ADSP-BF52x Blackfin® Processor Hardware Reference

Revision 1.0, March 2010

Part Number
82-000525

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

ANALOG DEVICES

Copyright Information

© 2010 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices logo, Blackfin, EZ-KIT Lite, SHARC, TigerSHARC, and VisualDSP++ are registered trademarks of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

CONTENTS

PREFACE

Purpose of This Manual lix
Intended Audience lix
What's New in This Manual lx
Technical or Customer Support lx
Product Information lxi
Analog Devices Web Site lxi
VisualDSP++ Online Documentation lxi
Technical Library CD lxii
Social Networking Web Sites lxiii
Notation Conventions lxiii

INTRODUCTION

Manual Contents 1-1
Peripherals 1-5
Memory Architecture 1-7
Internal Memory 1-9
External Memory 1-9
I/O Memory Space 1-9

One-Time-Programmable (OTP) Memory 1-10
DMA Support 1-11
External Bus Interface Unit 1-12
SDRAM Controller 1-12
Asynchronous Controller 1-13
Ports 1-13
General-Purpose I/O (GPIO) 1-13
Two-Wire Interface 1-15
Ethernet MAC 1-16
Parallel Peripheral Interface 1-16
SPORT Controllers 1-18
Serial Peripheral Interface (SPI) Port 1-20
Timers 1-20
UART Ports 1-21
Security 1-22
Real-Time Clock 1-23
Watchdog Timer 1-24
Clock Signals 1-25
Dynamic Power Management 1-26
Full-On Mode (Maximum Performance) 1-26
Active Mode (Moderate Power Savings) 1-26
Sleep Mode (High Power Savings) 1-26
Deep Sleep Mode (Maximum Power Savings) 1-27
Hibernate State 1-27

Voltage Regulation 1-27
Instruction Set Description 1-28
Development Tools 1-29

CHIP BUS HIERARCHY

Overview 2-1
Interface Overview 2-3
Internal Clocks 2-3
Core Bus Overview 2-4
Peripheral Access Bus (PAB) 2-6
PAB Arbitration 2-6
PAB Agents (Masters, Slaves) 2-6
PAB Performance 2-7
DMA Access Bus (DAB), DMA Core Bus (DCB), DMA External Bus (DEB) 2-8
DAB, DCB, DEB Arbitration 2-8
DCB Sharing 2-9
Using the CDPRIO Bit to Change Priorities 2-13
DAB Bus Agents (Masters) 2-13
DAB, DCB, and DEB Performance 2-14
External Access Bus (EAB) 2-14
Arbitration of the External Bus 2-15
DEB/EAB Performance 2-15

MEMORY

Memory Architecture 3-1

L1 Instruction SRAM 3-3
L1 Data SRAM 3-4
L1 Data Cache 3-4
Boot ROM 3-5
External Memory 3-5
Processor-Specific MMRs 3-5
DMEM_CONTROL Register 3-6
DTEST_COMMAND Register 3-6

ONE-TIME PROGRAMMABLE MEMORY

OTP Memory Overview 4-1
OTP Memory Map 4-2
Error Correction 4-7
Error Correction Policy 4-8
OTP Access 4-10
OTP Timing Parameters 4-12
Timing for the ADSP-BF523/525/527 Processors 4-13
Timing for the ADSP-BF522/524/526 Processors 4-14
OTP_TIMING Register 4-17
Callable ROM Functions for OTP ACCESS 4-17
Initializing OTP 4-17
bfrom_OtpCommand 4-17
Programming and Reading OTP 4-19
bfrom_OtpRead 4-20
bfrom_OtpWrite 4-21

Error Codes 4-24
Write-protecting OTP Memory 4-25
Accessing Private OTP Memory 4-28
OTP Programming Examples 4-28

SYSTEM INTERRUPTS

Specific Information for the ADSP-BF52x 5-1
Overview 5-1
Features 5-2
Description of Operation 5-2
Events and Sequencing 5-2
System Peripheral Interrupts 5-4
Programming Model 5-7
System Interrupt Initialization 5-8
System Interrupt Processing Summary 5-8
System Interrupt Controller Registers 5-10
System Interrupt Assignment (SIC_IAR) Register 5-11
System Interrupt Mask (SIC_IMASK) Register 5-12
System Interrupt Status (SIC_ISR) Register 5-12
System Interrupt Wakeup-Enable (SIC_IWR) Register 5-12
Programming Examples 5-13
Clearing Interrupt Requests 5-13
Unique Behavior for the ADSP-BF52x Processor 5-15
Interfaces 5-15
System Peripheral Interrupts 5-18

DIRECT MEMORY ACCESS

Specific Information for the ADSP-BF52x 6-1
Overview and Features 6-2
DMA Controller Overview 6-4
External Interfaces 6-4
Internal Interfaces 6-5
Peripheral DMA 6-6
Memory DMA 6-7
Handshaked Memory DMA (HMDMA) Mode 6-9
Modes of Operation 6-10
Register-Based DMA Operation 6-10
Stop Mode 6-11
Autobuffer Mode 6-12
Two-Dimensional DMA Operation 6-12
Examples of Two-Dimensional DMA 6-13
Descriptor-based DMA Operation 6-14
Descriptor List Mode 6-15
Descriptor Array Mode 6-16
Variable Descriptor Size 6-16
Mixing Flow Modes 6-17
Functional Description 6-18
DMA Operation Flow 6-18
DMA Startup 6-18
DMA Refresh 6-23

Work Unit Transitions 6-25
DMA Transmit and MDMA Source 6-26
DMA Receive 6-27
Stopping DMA Transfers 6-29
DMA Errors (Aborts) 6-29
DMA Control Commands 6-32
Restrictions 6-35
Transmit Restart or Finish 6-35
Receive Restart or Finish 6-36
Handshaked Memory DMA Operation 6-37
Pipelining DMA Requests 6-38
HMDMA Interrupts 6-41
DMA Performance 6-42
DMA Throughput 6-43
Memory DMA Timing Details 6-45
Static Channel Prioritization 6-46
Temporary DMA Urgency 6-46
Memory DMA Priority and Scheduling 6-48
Traffic Control 6-49
Programming Model 6-51
Synchronization of Software and DMA 6-52
Single-Buffer DMA Transfers 6-54
Continuous Transfers Using Autobuffering 6-54
Descriptor Structures 6-57

Descriptor Queue Management 6-58
Descriptor Queue Using Interrupts on Every Descriptor 6-58
Descriptor Queue Using Minimal Interrupts 6-60
Software Triggered Descriptor Fetches 6-62
DMA Registers 6-64
DMA Channel Registers 6-64
DMA Peripheral Map Registers(DMAx_PERIPHERAL_MAP/ MDMA_yy_PERIPHERAL_MAP) 6-68
DMA Configuration Registers (DMAx_CONFIG/MDMA_yy_CONFIG) 6-68
DMA Interrupt Status Registers (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS) 6-73
DMA Start Address Registers (DMAx_START_ADDR/MDMA_yy_START_ADDR) . 6-76
DMA Current Address Registers (DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR) ... 6-76
DMA Inner Loop Count Registers (DMAx_X_COUNT/MDMA_yy_X_COUNT) 6-77
DMA Current Inner Loop Count Registers (DMAx_CURR_X_COUNT /MDMA_yy_CURR_X_COUNT) 6-78
DMA Inner Loop Address Increment Registers (DMAx_X_MODIFY/MDMA_yy_X_MODIFY) 6-79
DMA Outer Loop Count Registers (DMAx_Y_COUNT/MDMA_yy_Y_COUNT) 6-80
DMA Current Outer Loop Count Registers (DMAx_CURR_Y_COUNT/ MDMA_yy_CURR_Y_COUNT) 6-81

DMA Outer Loop Address Increment Registers (DMAx_Y_MODIFY/MDMA_yy_Y_MODIFY) 6-81
DMA Next Descriptor Pointer Registers (DMAx_NEXT_DESC_PTR/ MDMA_yy_NEXT_DESC_PTR) 6-82
DMA Current Descriptor Pointer Registers (DMAx_CURR_DESC_PTR/ MDMA_yy_CURR_DESC_PTR) 6-83
HMDMA Registers 6-84
Handshake MDMA Control Registers (HMDMAx_CONTROL) 6-84
Handshake MDMA Initial Block Count Registers (HMDMAx_BCINIT) 6-87
Handshake MDMA Current Block Count Registers (HMDMAx_BCOUNT) 6-87
Handshake MDMA Current Edge Count Registers (HMDMAx_ECOUNT) 6-88
Handshake MDMA Initial Edge Count Registers (HMDMAx_ECINIT) 6-89
Handshake MDMA Edge Count Urgent Registers (HMDMAx_ECURGENT) 6-89
Handshake MDMA Edge Count Overflow Interrupt Registers (HMDMAx_ECOVERFLOW) 6-90
DMA Traffic Control Registers (DMA_TC_PER and DMA_TC_CNT) 6-90
DMA_TC_PER Register 6-91
DMA_TC_CNT Register 6-92
Programming Examples 6-93

Register-Based 2-D Memory DMA 6-94
Initializing Descriptors in Memory 6-97
Software-Triggered Descriptor Fetch Example 6-100
Handshaked Memory DMA Example 6-102
Unique Behavior for the ADSP-BF52x Processor 6-105
Static Channel Prioritization 6-106
DMA Control Commands 6-107
Handshaked Memory DMA Operation 6-107
HMDMA Interrupts 6-107

EXTERNAL BUS INTERFACE UNIT

EBIU Overview 7-1
Block Diagram 7-4
Internal Memory Interfaces 7-5
Registers 7-6
Shared Pins 7-6
System Clock 7-7
Error Detection 7-7
AMC Overview and Features 7-7
Features 7-8
Asynchronous Memory Interface 7-8
Asynchronous Memory Address Decode 7-9
AMC Pin Description 7-9
AMC Description of Operation 7-10
Avoiding Bus Contention 7-10

External Access Extension 7-11
AMC Functional Description 7-11
Programmable Timing Characteristics 7-11
Asynchronous Reads 7-11
Asynchronous Writes 7-13
Adding External Access Extension 7-15
Byte Enables 7-17
AMC Programming Model 7-17
AMC Registers 7-19
EBIU_AMGCTL Register 7-20
EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers 7-20
AMC Programming Examples 7-23
SDC Overview and Features 7-24
Features 7-24
SDRAM Configurations Supported 7-25
SDRAM External Bank Size 7-26
SDC Address Mapping 7-26
Internal SDRAM Bank Select 7-27
Parallel Connection of SDRAMs 7-28
SDC Interface Overview 7-28
SDC Pin Description 7-29
SDRAM Performance 7-30
SDC Description of Operation 7-31
Definition of SDRAM Architecture Terms 7-31

Refresh 7-31
Row Activation 7-31
Column Read/Write 7-31
Row Precharge 7-31
Internal Bank 7-32
External Bank 7-32
Memory Size 7-32
Burst Length 7-32
Burst Type 7-32
CAS Latency 7-33
Data I/O Mask Function 7-33
SDRAM Commands 7-33
Mode Register Set (MRS) command 7-33
Extended Mode Register Set (EMRS) command 7-33
Bank Activate command 7-33
Read/Write command 7-34
Precharge/Precharge All Command 7-34
Auto-refresh command 7-34
Enter Self-Refresh Mode 7-34
Exit Self-Refresh Mode 7-34
SDC Timing Specifications 7-35
t_{MRD} 7-35
t_{RAS} 7-35
t_{CL} 7-35

t_{RCD} 7-36
t_{RRD} 7-36
t_{WR} 7-36
t_{RP} 7-36
t_{RC} 7-37
t_{RFC} 7-37
t_{XSR} 7-37
t_{REF} 7-37
t_{REFI} 7-38
SDC Functional Description 7-38
SDC Operation 7-38
SDC Address Muxing 7-41
Multibank Operation 7-42
Core and DMA Arbitration 7-43
Changing System Clock During Runtime 7-44
Changing Power Management During Runtime 7-45
Deep Sleep Mode 7-45
Hibernate State 7-45
SDC Commands 7-46
Mode Register Set Command 7-47
Extended Mode Register Set Command (Mobile SDRAM) 7-48
Bank Activation Command 7-49
Read/Write Command 7-49

Write Command With Data Mask 7-50
Single Precharge Command 7-51
Precharge All Command 7-51
Auto-Refresh Command 7-51
Self-Refresh Mode 7-52
Self-Refresh Entry Command 7-52
Self-Refresh Exit Command 7-52
No Operation Command 7-53
SDC SA10 Pin 7-54
SDC Programming Model 7-54
SDC Configuration 7-54
Example SDRAM System Block Diagrams 7-56
SDC Register Definitions 7-59
EBIU_SDRRC Register 7-59
EBIU_SDBCTL Register 7-61
Using SDRAMs With Systems Smaller than 16M byte 7-63
EBIU_SDGCTL Register 7-65
SDRAM clock enable (SCTLE) 7-65
CAS latency (CL) 7-67
Partial array self refresh (PASR) 7-67
Bank activate command delay (TRAS) 7-68
Bank precharge delay (TRP) 7-68
RAS to CAS delay (TRCD) 7-68
Write to precharge delay (TWR) 7-69

Power-Up Start Delay (PUPSD) 7-69
Power-Up Sequence Mode (PSM) 7-70
Power-Up Sequence Start Enable (PSSE) 7-70
Self-Refresh Setting (SRFS) 7-71
Enter Self-Refresh Mode 7-71
Exit Self-Refresh Mode 7-72
External buffering enabled (EBUFE) 7-72
Fast Back-to-Back Read to Write (FBBRW) 7-73
Extended Mode Register Enabled (EMREN) 7-73
Temperature Compensated Self-Refresh (TCSR) 7-74
EBIU_SDSTAT Register 7-74
SDC Programming Examples 7-76

HOST DMA PORT

Overview 8-1
Features 8-2
Interface Overview 8-3
Description of Operation 8-3
Architecture 8-4
Functional Description 8-5
HOSTDP Configuration 8-5
HOSTDP Transactions 8-7
Host Read Status 8-8
Host Read Data and Host Write Data Operations 8-9
HOSTDP Modes of Operation 8-10

Acknowledge Mode 8-11
Acknowledge Mode Timing Diagrams 8-11
Host Bus Timeout 8-13
Interrupt Mode 8-14
DMA STOP Mode and AUTOBUFFER Mode 8-16
Bus Widths and Endian Order 8-16
Access Control 8-17
Improving HOSTDP DMA Bus Bandwidth 8-18
Control Commands Between the External Host and HOSTDP 8-19
Programming Model 8-21
Host DMA Port Registers 8-26
HOSTDP Control (HOST_CONTROL) Register 8-26
HOSTDP Status (HOST_STATUS) Register 8-28
HOSTDP Timeout (HOST_TIMEOUT) Register 8-31
Programming Examples 8-32

GENERAL-PURPOSE PORTS

Overview 9-1
Features 9-2
Interface Overview 9-3
External Interface 9-4
Port F Structure 9-4
Port G Structure 9-5
Port H Structure 9-7

Port J Structure 9-9
Input Tap Considerations 9-9
Internal Interfaces 9-10
Internal Signals 9-10
Performance/Throughput 9-11
Description of Operation 9-12
Operation 9-12
General-Purpose I/O Modules 9-13
GPIO Interrupt Processing 9-16
Programming Model 9-22
GPIO Drive Hysteresis Control 9-24
Portx Control (PORTx_HYSTERESIS) Register 9-24
Hysteresis Control Register 9-26
TWI Drive Strength Control Register 9-27
Memory-Mapped GPIO Registers 9-27
Port Multiplexer Control Register (PORTx_MUX) 9-28
Function Enable Registers (PORTx_FER) 9-30
GPIO Direction Registers (PORTxIO_DIR) 9-31
GPIO Input Enable Registers (PORTxIO_INEN) 9-32
GPIO Data Registers (PORTxIO) 9-32
GPIO Set Registers (PORTxIO_SET) 9-33
GPIO Clear Registers (PORTxIO_CLEAR) 9-33
GPIO Toggle Registers (PORTxIO_TOGGLE) 9-34
GPIO Polarity Registers (PORTxIO_POLAR) 9-34

Interrupt Sensitivity Registers (PORTxIO_EDGE) 9-35
GPIO Set on Both Edges Registers (PORTxIO_BOTH) 9-35
GPIO Mask Interrupt Registers (PORTxIO_MASKA/B) 9-36
GPIO Mask Interrupt Set Registers (PORTxIO_MASKA/B_SET) 9-37
GPIO Mask Interrupt Clear Registers (PORTxIO_MASKA/B_CLEAR) 9-39
GPIO Mask Interrupt Toggle Registers (PORTxIO_MASKA/B_TOGGLE) 9-41
Programming Examples 9-42

GENERAL-PURPOSE TIMERS

Specific Information for the ADSP-BF52x 10-1
Overview 10-2
External Interface 10-3
Internal Interface 10-4
Description of Operation 10-4
Interrupt Processing 10-5
Illegal States 10-7
Modes of Operation 10-10
Pulse Width Modulation (PWM_OUT) Mode 10-10
Output Pad Disable 10-12
Single Pulse Generation 10-12
Pulse Width Modulation Waveform Generation 10-13
PULSE_HI Toggle Mode 10-15
Externally Clocked PWM_OUT 10-20

Using PWM_OUT Mode With the PPI 10-21
Stopping the Timer in PWM_OUT Mode 10-21
Pulse Width Count and Capture (WDTH_CAP) Mode 10-23
Autobaud Mode 10-31
External Event (EXT_CLK) Mode 10-31
Programming Model 10-33
Timer Registers 10-34
Timer Enable Register (TIMER_ENABLE) 10-35
Timer Disable Register (TIMER_DISABLE) 10-36
Timer Status Register (TIMER_STATUS) 10-37
Timer Configuration Register (TIMER_CONFIG) 10-40
Timer Counter Register (TIMER_COUNTER) 10-41
Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers 10-43
Summary 10-46
Programming Examples 10-48
Unique Behavior for the ADSP-BF52x Processor 10-57
Interface Overview 10-58
External Interface 10-58

CORE TIMER

Specific Information for the ADSP-BF52x 11-1
Overview and Features 11-1
Timer Overview 11-2
External Interfaces 11-2

Internal Interfaces 11-3
Description of Operation 11-3
Interrupt Processing 11-3
Core Timer Registers 11-4
Core Timer Control Register (TCNTL) 11-5
Core Timer Count Register (TCOUNT) 11-5
Core Timer Period Register (TPERIOD) 11-6
Core Timer Scale Register (TSCALE) 11-7
Programming Examples 11-7
Unique Behavior for the ADSP-BF52x Processor 11-9

WATCHDOG TIMER

Specific Information for the ADSP-BF52x 12-1
Overview and Features 12-1
Interface Overview 12-3
External Interface 12-3
Internal Interface 12-3
Description of Operation 12-4
Register Definitions 12-5
Watchdog Count (WDOG_CNT) Register 12-5
Watchdog Status (WDOG_STAT) Register 12-6
Watchdog Control (WDOG_CTL) Register 12-7
Programming Examples 12-8
Unique Information for the ADSP-BF52x Processor 12-11

GENERAL-PURPOSE COUNTER

Specific Information for the ADSP-BF52x 13-1
Overview 13-2
Features 13-2
Interface Overview 13-3
Description of Operation 13-4
Quadrature Encoder Mode 13-4
Binary Encoder Mode 13-5
Up/Down Counter Mode 13-6
Direction Counter Mode 13-7
Timed Direction Mode 13-7
Functional Description 13-7
Input Noise Filtering (Debouncing) 13-8
Zero Marker (Push Button) Operation 13-9
Boundary Comparison Modes 13-10
Control and Signaling Events 13-12
Illegal Gray/Binary Code Events 13-12
Up/Down Count Events 13-12
Zero-Count Events 13-13
Overflow Events 13-13
Boundary Match Events 13-13
Zero Marker Events 13-14
Capturing Timing Information 13-14

Capturing Time Interval Between Successive Counter Events 13-15
Capturing Counter Interval and CNT_COUNTER Read Timing 13-16
Programming Model 13-18
Registers 13-19
Counter Module Register Overview 13-19
Counter Configuration Register (CNT_CONFIG) 13-20
Counter Interrupt Mask Register (CNT_IMASK) 13-20
Counter Status Register (CNT_STATUS) 13-21
Counter Command Register (CNT_COMMAND) 13-22
Counter Debounce Register (CNT_DEBOUNCE) 13-24
Counter Count Value Register (CNT_COUNTER) 13-25
Counter Boundary Registers (CNT_MIN and CNT_MAX) . 13-26
Programming Examples 13-27
Unique Behavior for the ADSP-BF52x Processor 13-38

REAL-TIME CLOCK

Specific Information for the ADSP-BF52x 14-1
Overview 14-1
Interface Overview 14-3
Description of Operation 14-4
RTC Clock Requirements 14-5
Prescaler Enable 14-5
RTC Programming Model 14-7

Register Writes 14-8
Write Latency 14-9
Register Reads 14-10
Deep Sleep 14-10
Event Flags 14-11
Setting Time of Day 14-13
Using the Stopwatch 14-13
Interrupts 14-14
State Transitions Summary 14-16
Register Definitions 14-19
RTC Status (RTC_STAT) Register 14-20
RTC Interrupt Control (RTC_ICTL) Register 14-20
RTC Interrupt Status (RTC_ISTAT) Register 14-21
RTC Stopwatch Count (RTC_SWCNT) Register 14-21
RTC Alarm (RTC_ALARM) Register 14-22
RTC Prescaler Enable (RTC_PREN) Register 14-22
Programming Examples 14-22
Enable RTC Prescaler 14-23
RTC Stopwatch For Exiting Deep Sleep Mode 14-23
RTC Alarm to Come Out of Hibernate State 14-25
Unique Information for the ADSP-BF52x Processor 14-27

PARALLEL PERIPHERAL INTERFACE

Specific Information for the ADSP-BF52x 15-1
Overview 15-2

Features 15-2
Interface Overview 15-3
Description of Operation 15-4
Functional Description 15-5
ITU-R 656 Modes 15-5
ITU-R 656 Background 15-5
ITU-R 656 Input Modes 15-9
Entire Field 15-9
Active Video Only 15-10
Vertical Blanking Interval (VBI) only 15-10
ITU-R 656 Output Mode 15-11
Frame Synchronization in ITU-R 656 Modes 15-11
General-Purpose PPI Modes 15-12
Data Input (RX) Modes 15-14
No Frame Syncs 15-15
1, 2, or 3 External Frame Syncs 15-16
2 or 3 Internal Frame Syncs 15-16
Data Output (TX) Modes 15-17
No Frame Syncs 15-17
1 or 2 External Frame Syncs 15-18
1, 2, or 3 Internal Frame Syncs 15-19
Frame Synchronization in GP Modes 15-20
Modes With Internal Frame Syncs 15-20
Modes With External Frame Syncs 15-21

Programming Model 15-22
DMA Operation 15-23
PPI Registers 15-26
PPI Control Register (PPI_CONTROL) 15-26
PPI Status Register (PPI_STATUS) 15-31
PPI Delay Count Register (PPI_DELAY) 15-34
PPI Transfer Count Register (PPI_COUNT) 15-34
PPI Lines Per Frame Register (PPI_FRAME) 15-35
Programming Examples 15-37
Unique Behavior for the ADSP-BF52x Processor 15-39

SECURITY

Overview 16-1
Features 16-4
Description of Operation 16-6
Secure State Machine 16-7
Open Mode 16-8
Secure Entry Mode 16-8
Secure Mode 16-9
SecureMode Control 16-11
Security Features 16-13
Digital Signature Authentication 16-13
Digital Signature Authentication Performance Measurement 16-16
Protection Features 16-16

Operating in Secure Mode 16-20
Entering Secure Mode 16-20
Exiting Secure Mode 16-21
Reset Handling in Secure Mode 16-21
Hardware Reset 16-21
Clearing Private Data 16-22
Public Key Requirements 16-24
Storing Public Cipher Key in Public OTP 16-26
Cryptographic Ciphers 16-27
Keys 16-27
Debug Functionality 16-27
Programming Examples 16-31
Programming Model 16-32
Secure Entry Service Routine (SESR) API 16-32
Starting Authentication 16-33
Memory Configuration 16-34
Message Placement 16-35
Digital Signature 16-35
Message Size Constraints 16-35
Memory Usage 16-36
Memory Protection 16-36
Secure Function and Secure Entry Service Routine Arguments 16-36
Secure Function Arguments 16-37
Secure Entry Service Routine Arguments 16-38

usFlags 16-38
uslRQMask 16-39
ulMessageSize 16-40
ulSFEntryPoint 16-40
ulMessagePtr 16-40
Secure Message Execution 16-40
Return Codes 16-41
Secure Hash Algorithm (SHA-1) API 16-43
ADI_SHA1 Data Type 16-43
bfrom_Sha1Init ROM Routine 16-44
bfrom_Sha1Hash ROM Routine 16-44
Security Registers 16-45
Secure System Switch (SECURE_SYSSWT) Register 16-46
Secure Control (SECURE_CONTROL) Register 16-52
Secure Status (SECURE_STATUS) Register 16-55

SYSTEM RESET AND BOOTING

Overview 17-1
Reset and Power-up 17-4
Hardware Reset 17-6
Software Resets 17-7
Reset Vector 17-8
Servicing Reset Interrupts 17-10
Preboot 17-11
Factory Page Settings (FPS) 17-14

Preboot Page Settings (PBS) 17-14
Alternative PBS Pages 17-16
Programming PBS Pages 17-16
Recovering From Misprogrammed PBS Pages 17-17
Customizing Power Management 17-17
Customizing Booting Options 17-18
Customizing the Asynchronous Port 17-19
Customizing the Synchronous Port 17-20
Basic Booting Process 17-21
Block Headers 17-23
Block Code 17-25
DMA Code Field 17-25
Block Flags Field 17-27
Header Checksum Field 17-28
Header Sign Field 17-29
Target Address 17-29
Byte Count 17-30
Argument 17-31
Boot Host Wait (HWAIT) Feedback Strobe 17-31
Using HWAIT as Reset Indicator 17-32
Boot Termination 17-33
Single Block Boot Streams 17-34
Direct Code Execution 17-34
Advanced Boot Techniques 17-37

Initialization Code 17-37
Quick Boot 17-41
Indirect Booting 17-42
Callback Routines 17-43
Error Handler 17-46
CRC Checksum Calculation 17-46
Load Functions 17-47
Calling the Boot Kernel at Runtime 17-48
Debugging the Boot Process 17-49
Boot Management 17-51
Booting a Different Application 17-52
Multi-DXE Boot Streams 17-53
Determining Boot Stream Start Addresses 17-57
Initialization Hook Routine 17-57
Specific Boot Modes 17-58
No Boot Mode 17-59
Flash Boot Modes 17-59
SDRAM Boot Mode 17-63
FIFO Boot Mode 17-63
SPI Master Boot Modes 17-65
SPI Device Detection Routine 17-67
SPI Slave Boot Mode 17-69
TWI Master Boot Mode 17-72
TWI Slave Boot Mode 17-75

UART Slave Mode Boot 17-78
OTP Boot Mode 17-80
Host DMA Boot Modes 17-81
NAND Flash Boot Mode 17-86
Supported Devices 17-86
NAND Flash Page Structure 17-89
Auto Detection 17-91
Boot Stream Processing 17-91
Software Configurable NAND Flash Boot Modes 17-93
Sequential Block Mode 17-94
Block Skip Mode 17-95
Multiple Image Mode 17-96
Reset and Booting Registers 17-98
Software Reset (SWRST) Register 17-99
System Reset Configuration (SYSCR) Register 17-100
Boot Code Revision Control (BK_REVISION) 17-103
Boot Code Date Code (BK_DATECODE) 17-104
Zero Word (BK_ZEROS) 17-105
Ones Word (BK_ONES) 17-106
OTP Memory Pages for Booting 17-106
Lower PBS00 Half Page 17-106
Upper PBS00 Half Page 17-110
Lower PBS01 Half Page 17-111
Upper PBS01 Half Page 17-111

Lower PBS02 Half Page 17-114
Upper PBS02 Half Page 17-115
Reserved Half Pages 17-115
Data Structures 17-115
ADI_BOOT_HEADER 17-115
ADI_BOOT_BUFFER 17-116
ADI_BOOT_DATA 17-116
dFlags Word 17-120
ADI_BOOT_NAND 17-121
ADI_BOOT_NAND_DEVICE 17-122
ADI_BOOT_NAND_BUFFER 17-124
ADI_BOOT_NAND_ACCESS 17-125
ADI_BOOT_NAND_ADDRESS 17-125
ADI_BOOT_NAND_ECC 17-127
Callable ROM Functions for Booting 17-129
BFROM_FINALINIT 17-129
BFROM_PDMA 17-129
BFROM_MDMA 17-130
BFROM_MEMBOOT 17-130
BFROM_TWIBOOT 17-132
BFROM_SPIBOOT 17-132
BFROM_OTPBOOT 17-133
BFROM_NANDBOOT 17-134
BFROM_BOOTKERNEL 17-135

BFROM_CRC32 17-136
BFROM_CRC32POLY 17-136
BFROM_CRC32CALLBACK 17-137
BFROM_CRC32INITCODE 17-137
Programming Examples 17-138
System Reset 17-138
Exiting Reset to User Mode 17-139
Exiting Reset to Supervisor Mode 17-139
Initcode (SDRAM Controller Setup) 17-140
Initcode (Power Management Control) 17-142
Initcode (NAND Flash Boot Mode Configuration) 17-144
Quickboot With Restore From SDRAM 17-145
XOR Checksum 17-146
Direct Code Execution 17-148
Managing PBS Pages in OTP Memory 17-149

DYNAMIC POWER MANAGEMENT

Phase Locked Loop and Clock Control 18-1
PLL Overview 18-2
PLL Clock Multiplier Ratios 18-3
Core Clock/System Clock Ratio Control 18-5
Dynamic Power Management Controller 18-7
Operating Modes 18-8
Dynamic Power Management Controller States 18-8
Full-On Mode 18-8

Active Mode 18-9
Sleep Mode 18-9
Deep Sleep Mode 18-10
Hibernate State 18-11
Operating Mode Transitions 18-11
Programming Operating Mode Transitions 18-14
Dynamic Supply Voltage Control 18-16
Power Supply Management 18-17
Controlling the Internal Voltage Regulator 18-19
Changing Voltage on ADSP-BF523/ADSP-BF525/ADSP-BF527 18-19
Changing Voltage on ADSP-BF522/ADSP-BF524/ADSP-BF526 18-21
Powering Down the Core (Hibernate State) 18-22
PLL and VR Registers 18-25
PLL_DIV Register 18-26
PLL_CTL Register 18-26
PLL_STAT Register 18-28
PLL_LOCKCNT Register 18-28
VR_CTL Register 18-28
System Control ROM Function 18-30
Programming Model 18-32
Accessing the System Control ROM Function in C/C++ 18-32
Accessing the System Control ROM Function in Assembly ... 18-33
Programming Examples 18-36

Full-on Mode to Active Mode and Back 18-37
Transition to Sleep Mode or Deep Sleep Mode 18-39
Set Wakeups and Entering Hibernate State 18-40
Perform a System Reset or Soft-Reset 18-42
In Full-on Mode, Change VCO Frequency, Core Clock Frequency, and System Clock Frequency 18-43
Changing Voltage Levels 18-45

SYSTEM DESIGN

Pin Descriptions 19-1
Managing Clocks 19-1
Managing Core and System Clocks 19-2
Configuring and Servicing Interrupts 19-2
Semaphores 19-2
Example Code for Query Semaphore 19-3
Data Delays, Latencies and Throughput 19-4
Bus Priorities 19-4
External Memory Design Issues 19-4
Example Asynchronous Memory Interfaces 19-5
Avoiding Bus Contention 19-7
High-Frequency Design Considerations 19-7
Signal Integrity 19-8
Decoupling Capacitors and Ground Planes 19-9
5 Volt Tolerance 19-11

Test Point Access 19-11
Oscilloscope Probes 19-11
Recommended Reading 19-11
Resetting the Processor 19-13
Recommendations for Unused Pins 19-13
Programmable Outputs 19-13
USB System Hardware Design 19-14
Voltage Regulator System Hardware Design 19-15
For VRSEL = Logic 0, Internal Regulator, SS Mode 19-16
For VRSEL = Logic 1, External Regulator, PG Mode 19-16

NAND FLASH CONTROLLER

Overview 20-1
Features 20-2
Interface Overview 20-3
Description of Operation 20-3
Internal Bus Interfaces 20-3
Bus Access Types 20-4
Access Timing 20-5
Functional Description 20-6
Page Write 20-6
Page Read 20-7
Additional Operations 20-8
Write Protection 20-9
Chip Enable Don't Care 20-9

NFC Error Detection 20-9
Error Analysis 20-10
Large Page Size Support 20-12
NFC SmartMedia Support 20-12
Programming Model 20-13
NFC Registers 20-14
NFC Control (NFC_CTL) Register 20-16
NFC Status (NFC_STAT) Register 20-16
NFC Interrupt Status (NFC_IRQSTAT) Register 20-17
NFC Interrupt Mask (NFC_IRQMASK) Register 20-19
NFC ECC (NFC_ECCx) Registers 20-19
NFC Count (NFC_COUNT) Register 20-20
NFC Reset (NFC_RST) Register 20-21
NFC Page Control (NFC_PGCTL) Register 20-21
NFC Read Data (NFC_READ) Register 20-22
NFC Address (NFC_ADDR) Register 20-23
NFC Command (NFC_CMD) Register 20-23
NFC Data Write (NFC_DATA_WR) Register 20-24
NFC Data Read (NFC_DATA_RD) Register 20-25
NFC Programming Examples 20-25

ETHERNET MAC

Specific Information for the ADSP-BF52x 21-1
Overview 21-2
Features 21-2

Interface Overview 21-3
External Interface 21-4
Clocking 21-4
Pins 21-5
Internal Interface 21-7
Power Management 21-7
Description of Operation 21-7
Protocol 21-8
MII Management Interface 21-8
Operation 21-10
MII Management Interface Operation 21-10
Receive DMA Operation 21-11
Frame Reception and Filtering 21-13
Discarded Frames 21-15
Aborted Frames 21-16
Control Frames 21-16
Examples 21-16
RX Automatic Pad Stripping 21-17
RX DMA Data Alignment 21-17
RX DMA Buffer Structure 21-18
RX Frame Status Buffer 21-19
RX Frame Status Classification 21-19
RX IP Frame Checksum Calculation 21-21
RX DMA Direction Errors 21-22

Transmit DMA Operation 21-23
Flexible Descriptor Structure 21-26
TX DMA Data Alignment 21-27
Late Collisions 21-28
TX Frame Status Classification 21-28
TX DMA Direction Errors 21-29
Power Management 21-30
Ethernet Operation in the Sleep State 21-32
Magic Packet Detection 21-34
Remote Wake-up Filters 21-34
Ethernet Event Interrupts 21-39
RX/TX Frame Status Interrupt Operation 21-42
RX Frame Status Register Operation at Startup and Shutdown 21-42
TX Frame Status Register Operation at Startup and Shutdown 21-43
MAC Management Counters 21-43
Programming Model 21-46
Configure MAC Pins 21-46
Multiplexing Scheme 21-46
CLKBUF 21-47
Configure Interrupts 21-47
Configure MAC Registers 21-48
MAC Address 21-48
MII Station Management 21-48

Configure PHY 21-49
Receive and Transmit Data 21-50
Receiving Data 21-50
Transmitting Data 21-51
Ethernet MAC Register Definitions 21-51
Control-Status Register Group 21-60
MAC Operating Mode (EMAC_OPMODE) Register 21-61
MAC Address Low (EMAC_ADDRLO) Register 21-67
MAC Address High Register (EMAC_ADDRHI) Register 21-68
MAC Multicast Hash Table High (EMAC_HASHHI) and Low (EMAC_HASHLO) Registers 21-69
MAC Station Management Address (EMAC_STAADD) Register 21-72
MAC Station Management Data (EMAC_STADAT) Register 21-74
MAC Flow Control (EMAC_FLC) Register 21-75
MAC VLAN1 Tag (EMAC_VLAN1) and MAC VLAN2 Tag (EMAC_VLAN2)Registers 21-77
MAC Wakeup Frame Control and Status (EMAC_WKUP_CTL) Register 21-78
MAC Wakeup Frame0 Byte Mask (EMAC_WKUP_FFMSK0)
MAC Wakeup Frame1 Byte Mask (EMAC_WKUP_FFMSK1)
MAC Wakeup Frame2 Byte Mask (EMAC_WKUP_FFMSK2)
MAC Wakeup Frame3 Byte Mask (EMAC_WKUP_FFMSK3) Registers 21-81
MAC Wakeup Frame Filter Commands (EMAC_WKUP_FFCMD) Register 21-86

Ethernet MAC Wakeup Frame Filter Offsets (EMAC_WKUP_FFOFF) Register 21-88

MAC Wakeup Frame Filter CRC0/1 (EMAC_WKUP_FFCRC0) and CRC2/3 (EMAC_WKUP_FFCRC1) Registers 21-88

System Interface Register Group 21-89

MAC System Control (EMAC_SYSCTL) Register 21-90

MAC System Status (EMAC_SYSTAT) Register 21-91

Ethernet MAC Frame Status Registers 21-94

Ethernet MAC RX Current Frame Status (EMAC_RX_STAT) Register 21-94

Ethernet MAC RX Sticky Frame Status (EMAC_RX_STKY) Register 21-100

Ethernet MAC RX Frame Status Interrupt Enable (EMAC_RX_IRQE) Register 21-104

Ethernet MAC TX Current Frame Status (EMAC_TX_STAT) Register 21-105

Ethernet MAC TX Sticky Frame Status (EMAC_TX_STKY) Register 21-109

Ethernet MAC TX Frame Status Interrupt Enable (EMAC_TX_IRQE) Register 21-112

Ethernet MAC MMC RX Interrupt Status (EMAC_MMC_RIRQS) Register 21-112

Ethernet MAC MMC RX Interrupt Enable (EMAC_MMC_RIRQE) Register 21-114

Ethernet MAC MMC TX Interrupt Status (EMAC_MMC_TIRQS) Register 21-116

Ethernet MAC MMC TX Interrupt Enable (EMAC_MMC_TIRQE) Register 21-118

MAC Management Counter Registers 21-120
MAC Management Counters Control (EMAC_MMC_CTL) Register 21-121
Programming Examples 21-122
Ethernet Structures 21-123
MAC Address Setup 21-126
PHY Control Routines 21-126
Unique Behavior for the ADSP-BF52x Processor 21-128

SPI-COMPATIBLE PORT CONTROLLER

Specific Information for the ADSP-BF52x 22-1
Overview 22-2
Features 22-2
Interface Overview 22-3
External Interface 22-4
SPI Clock Signal (SCK) 22-4
Master-Out, Slave-In (MOSI) Signal 22-5
Master-In, Slave-Out (MISO) Signal 22-5
SPI Slave Select Input Signal (SPISS) 22-6
SPI Slave Select Enable Output Signals 22-7
Slave Select Inputs 22-8
Use of FLS Bits in SPI_FLG for Multiple Slave SPI Systems 22-8
Internal Interfaces 22-11
DMA Functionality 22-11
Description of Operation 22-12

SPI Transfer Protocols 22-12
SPI General Operation 22-15
Clock Signals 22-16
Interrupt Output 22-17
Functional Description 22-17
Master Mode Operation (Non-DMA) 22-18
Transfer Initiation From Master (Transfer Modes) 22-19
Slave Mode Operation (Non-DMA) 22-20
Slave Ready for a Transfer 22-22
Programming Model 22-22
Beginning and Ending an SPI Transfer 22-22
Master Mode DMA Operation 22-24
Slave Mode DMA Operation 22-27
SPI Registers 22-34
SPI Baud Rate (SPI_BAUD) Register 22-34
SPI Control (SPI_CTL) Register 22-35
SPI Flag (SPI_FLG) Register 22-38
SPI Status (SPI_STAT) Register 22-40
Mode Fault Error (MODF) 22-41
Transmission Error (TXE) 22-42
Reception Error (RBSY) 22-42
Transmit Collision Error (TXCOL) 22-42
SPI Transmit Data Buffer (SPI_TDBR) Register 22-42
SPI Receive Data Buffer (SPI_RDBR) Register 22-43

SPI RDBR Shadow (SPI_SHADOW) Register 22-44
Programming Examples 22-45
Core-Generated Transfer 22-45
Initialization Sequence 22-45
Starting a Transfer 22-46
Post Transfer and Next Transfer 22-47
Stopping 22-48
DMA-Based Transfer 22-48
DMA Initialization Sequence 22-48
SPI Initialization Sequence 22-49
Starting a Transfer 22-51
Stopping a Transfer 22-51
Unique Behavior for the ADSP-BF52x Processor 22-53

TWO WIRE INTERFACE CONTROLLER

Specific Information for the ADSP-BF52x 23-1
Overview 23-2
Interface Overview 23-3
External Interface 23-4
Serial Clock Signal (SCL) 23-4
Serial Data Signal (SDA) 23-4
TWI Pins 23-5
Internal Interfaces 23-5
Description of Operation 23-6
TWI Transfer Protocols 23-6

Clock Generation and Synchronization 23-7
Bus Arbitration 23-8
Start and Stop Conditions 23-9
General Call Support 23-10
Fast Mode 23-10
Functional Description 23-10
General Setup 23-11
Slave Mode 23-11
Master Mode Clock Setup 23-12
Master Mode Transmit 23-13
Master Mode Receive 23-14
Repeated Start Condition 23-15
Transmit/Receive Repeated Start Sequence 23-15
Receive/Transmit Repeated Start Sequence 23-17
Clock Stretching 23-18
Clock Stretching During FIFO Underflow 23-18
Clock Stretching During FIFO Overflow 23-20
Clock Stretching During Repeated Start Condition 23-21
Programming Model 23-24
Register Descriptions 23-26
TWI CONTROL Register (TWI_CONTROL) 23-26
SCL Clock Divider Register (TWI_CLKDIV) 23-27
TWI Slave Mode Control Register (TWI_SLAVE_CTL) 23-28
TWI Slave Mode Address Register (TWI_SLAVE_ADDR) .. 23-30

TWI Slave Mode Status Register (TWI_SLAVE_STAT) 23-30
TWI Master Mode Control Register (TWI_MASTER_CTL) 23-32
TWI Master Mode Address Register (TWI_MASTER_ADDR) 23-34
TWI Master Mode Status Register (TWI_MASTER_STAT) . 23-35
TWI FIFO Control Register (TWI_FIFO_CTL) 23-38
TWI FIFO Status Register (TWI_FIFO_STAT) 23-40
TWI FIFO Status 23-40
TWI Interrupt Mask Register (TWI_INT_MASK) 23-41
TWI Interrupt Status Register (TWI_INT_STAT) 23-42
TWI FIFO Transmit Data Single Byte Register (TWI_XMT_DATA8) 23-45
TWI FIFO Transmit Data Double Byte Register (TWI_XMT_DATA16) 23-45
TWI FIFO Receive Data Single Byte Register (TWI_RCV_DATA8) 23-46
TWI FIFO Receive Data Double Byte Register (TWI_RCV_DATA16) 23-47
Programming Examples 23-48
Master Mode Setup 23-48
Slave Mode Setup 23-53
Electrical Specifications 23-60
Unique Information for the ADSP-BF52x Processor 23-60

SPORT CONTROLLER

Specific Information for the ADSP-BF52x 24-1
Overview 24-2

Features 24-2
Interface Overview 24-4
SPORT Pin/Line Terminations 24-9
Description of Operation 24-10
SPORT Disable 24-10
Setting SPORT Modes 24-11
Stereo Serial Operation 24-11
Multichannel Operation 24-15
Multichannel Enable 24-18
Frame Syncs in Multichannel Mode 24-19
The Multichannel Frame 24-20
Multichannel Frame Delay 24-21
Window Size 24-21
Window Offset 24-22
Other Multichannel Fields in SPORT_MCMC2 24-22
Channel Selection Register 24-23
Multichannel DMA Data Packing 24-24
Support for H.100 Standard Protocol 24-25
2× Clock Recovery Control 24-25
Functional Description 24-26
Clock and Frame Sync Frequencies 24-26
Maximum Clock Rate Restrictions 24-27
Word Length 24-28
Bit Order 24-28

Data Type 24-29
Companding 24-29
Clock Signal Options 24-30
Frame Sync Options 24-31
Framed Versus Unframed 24-31
Internal Versus External Frame Syncs 24-33
Active Low Versus Active High Frame Syncs 24-34
Sampling Edge for Data and Frame Syncs 24-34
Early Versus Late Frame Syncs (Normal Versus Alternate Timing) 24-36
Data Independent Transmit Frame Sync 24-38
Moving Data Between SPORTs and Memory 24-39
SPORT RX, TX, and Error Interrupts 24-39
Peripheral Bus Errors 24-40
Timing Examples 24-40
SPORT Registers 24-46
Register Writes and Effective Latency 24-47
SPORT Transmit Configuration (SPORT_TCR1 and SPORT_TCR2) Registers 24-48
SPORT Receive Configuration (SPORT_RCR1 and SPORT_RCR2) Registers 24-54
Data Word Formats 24-58
SPORT Transmit Data (SPORT_TX) Register 24-59
SPORT Receive Data (SPORT_RX) Register 24-61
SPORT Status (SPORT_STAT) Register 24-64

SPORT Transmit and Receive Serial Clock Divider (SPORT_TCLKDIV and SPORT_RCLKDIV) Registers ... 24-65
SPORT Transmit and Receive Frame Sync Divider (SPORT_TFSDIV and SPORT_RFSDIV) Registers 24-66
SPORT Multichannel Configuration (SPORT_MCMC1 and SPORT_MCMC2) Registers 24-67
SPORT Current Channel (SPORT_CHNL) Register 24-68
SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers 24-69
SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers 24-70
Programming Examples 24-71
SPORT Initialization Sequence 24-72
DMA Initialization Sequence 24-74
Interrupt Servicing 24-76
Starting a Transfer 24-77
Unique Information for the ADSP-BF52x Processor 24-78

UART PORT CONTROLLERS

Specific Information for the ADSP-BF52x 25-1
Overview 25-2
Features 25-2
Interface Overview 25-3
External Interface 25-3
Internal Interface 25-4
Description of Operation 25-5

UART Transfer Protocol 25-5
UART Transmit Operation 25-6
UART Receive Operation 25-7
IrDA Transmit Operation 25-9
IrDA Receive Operation 25-9
Interrupt Processing 25-11
Bit Rate Generation 25-13
Autobaud Detection 25-14
Programming Model 25-16
Non-DMA Mode 25-16
DMA Mode 25-18
Mixing Modes 25-19
UART Registers 25-20
UART Line Control (UART_LCR) Register 25-22
UART Modem Control (UART_MCR) Register 25-24
UART Line Status (UART_LSR) Register 25-25
UART Transmit Holding (UART_THR) Register 25-26
UART Receive Buffer (UART_RBR) Register 25-27
UART Interrupt Enable (UART_IER) Register 25-27
UART Interrupt Identification (UART_IIR) Register 25-29
UART Divisor Latch (UART_DLL and UART_DLH) Registers 25-30
UART Scratch (UART_SCR) Register 25-32
UART Global Control (UART_GCTL) Register 25-32
Programming Examples 25-33

Unique Information for the ADSP-BF52x Processor 25-43

USB OTG CONTROLLER

Overview 26-1
Features 26-2
Interface Overview 26-3
FIFO Configuration 26-7
Interrupts 26-8
Resets 26-11
Description of Operation 26-12
Peripheral Mode Operation 26-12
Endpoint Setup 26-12
IN Transactions as a Peripheral 26-14
OUT Transactions as a Peripheral 26-15
Peripheral Transfer Workflows 26-16
Control Transactions as a Peripheral 26-18
Write Requests 26-18
Read Requests 26-20
Zero Data Requests 26-21
ENDPOINT 0 States 26-22
Endpoint 0 Service Routine as Peripheral 26-24
Idle Mode 26-27
TX Mode 26-27
RX Mode 26-29
Peripheral Mode, Bulk IN, Transfer Size Known 26-32

Peripheral Mode, Bulk IN, Transfer Size Unknown 26-32
Peripheral Mode, ISO IN, Small *MaxPktSize* 26-33
Peripheral Mode, ISO IN, Large *MaxPktSize* 26-34
Peripheral Mode, Bulk OUT, Transfer Size Known 26-35
Peripheral Mode, Bulk OUT, Transfer Size Unknown ... 26-35
Peripheral Mode, ISO OUT, Small *MaxPktSize* 26-36
Peripheral Mode, ISO OUT, Large *MaxPktSize* 26-37
Peripheral Mode Suspend 26-37
Start-of-frame (SOF) Packets 26-38
Soft Connect/Soft Disconnect 26-38
Error Handling As a Peripheral 26-39
Stalls Issued to Control Transfers 26-40
Zero Length OUT Data Packets in Control Transfers 26-41
Host Mode Operation 26-41
Endpoint Setup and Data Transfer 26-41
Control Transaction as a Host 26-42
Setup Phase as a Host 26-43
IN Data Phase as a Host 26-44
OUT Data as a Host (Control) 26-45
IN Status Phase as a Host (Following SETUP Phase or OUT Data Phase) 26-46
OUT Status Phase as a Host (following IN Data Phase) ... 26-47
Host IN Transactions 26-48
Host OUT Transactions 26-49
Transaction Scheduling 26-49

Babble 26-50
Host Mode Reset 26-51
Host Mode Suspend 26-51
Functional Description 26-51
On-Chip Bus Interfaces 26-51
Interface Pins 26-53
Power and Clocking 26-53
UTMI Interface 26-54
Programming Model 26-54
Peripheral Mode Flow Charts 26-55
Host Mode Flow Charts 26-64
DMA Mode Flow Charts 26-73
OTG Session Request 26-78
Starting a Session 26-78
Detecting Activity 26-79
Host Negotiation/Configuration 26-80
Software Clock Control 26-81
Wakeup from Hibernate State 26-81
Wakeup Without Re-Enumeration 26-83
Data Transfer 26-85
Loading/Unloading Packets from Endpoints 26-86
DMA Master Channels 26-87
DMA Bus Cycles 26-89
Transferring Packets Using DMA 26-90

Individual Packet: RX Endpoint 26-91
Individual Packet: TX Endpoint 26-92
Multiple Packets: RX Endpoint 26-92
Multiple Packets: TX Endpoints 26-94
USB OTG Registers 26-95
USB Global Control (USB_GLOBAL_CTL) Register 26-95
USB Power Management (USB_POWER) Register 26-98
USB Function Address (USB_FADDR) Register 26-100
USB Test Mode (USB_TESTMODE) Register 26-101
USB Global Interrupt (USB_GLOBINTR) Register 26-102
USB Transmit Interrupt (USB_INTRTX) Register 26-103
USB Receive Interrupt (USB_INTRRX) Register 26-103
USB Transmit Interrupt Enable (USB_INTRTXE) Register 26-105
USB Receive Interrupt Enable (USB_INTRRXE) Register .. 26-106
USB Common Interrupts (USB_INTRUSB) Register 26-107
USB Common Interrupt Enable (USB_INTRUSBE) Register 26-107
USB Frame Number (USB_FRAME) Register 26-109
USB Index (USB_INDEX) Register 26-109
USB TX Max Packet (USB_TX_MAX_PACKET) Register 26-110
USB Control/Status EP0 (USB_CSR0) Register 26-110
USB TX Control/Status EPx (USB_TXCSR) Register 26-115
USB RX Max Packet (USB_RX_MAX_PACKET) Register . 26-120
USB RX Control/Status (USB_RXCSR) Register 26-121
USB Count 0 (USB_COUNT0) Register 26-126

USB RX Byte Count EPx (USB_RXCOUNT) Register 26-126
USB TX Type (USB_TXTYPE) Register 26-127
USB NAK Limit 0 (USB_NAKLIMIT0) Register 26-128
USB TX Interval (USB_TXINTERVAL) Register 26-128
USB RX Type (USB_RXTYPE) Register 26-129
USB RX Interval (USB_RXINTERVAL) Register 26-130
USB TX Byte Count EPx (USB_TXCOUNT) Register 26-131
USB Endpoint FIFO (USB_EPx_FIFO) Registers 26-132
USB OTG Device Control (USB_OTG_DEV_CTL) Register 26-132
USB OTG VBUS Interrupt (USB_OTG_VBUS_IRQ) Register 26-134
USB OTG VBUS Mask (USB_OTG_VBUS_MASK) Register 26-136
USB Link Info (USB_LINKINFO) Register 26-137
USB VBUS Pulse Length (USB_VPLEN) Register 26-137
USB High-Speed EOF 1 (USB_HS_EOF1) Register 26-138
USB Full-Speed EOF 1 (USB_FS_EOF1) Register 26-138
USB Low-Speed EOF 1 (USB_LS_EOF1) Register 26-139
USB APHY Control 2 (USB_APHY_CNTRL2) Register ... 26-140
USB PLL OSC Control (USB_PLLOSC_CTRL) Registers 26-140
USB SRP Clock Divider (USB_SRP_CLKDIV) Register ... 26-142
USB DMA Interrupt (USB_DMA_INTERRUPT) Register 26-143
USB DMAx Control (USB_DMA_CONTROL) Registers . 26-143
USB DMAx Address Low (USB_DMAxADDRLOW) Registers 26-146
USB DMAx Address High (USB_DMAxADDRHIGH) Registers .. 26-146

USB DMAx Count Low (USB_DMAxCOUNTLOW) Registers 26-147

USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers . 26-147

References 26-148

Glossary of USB Terms 26-148

SYSTEM MMR ASSIGNMENTS

Dynamic Power Management Registers A-3

System Reset and Interrupt Control Registers A-4

OTP Memory Registers A-5

Watchdog Timer Registers A-5

Real-Time Clock Registers A-6

UART0 Controller Registers A-6

SPI Controller Registers A-7

Timer Registers A-8

Ports Registers A-9

SPORT0 Controller Registers A-12

SPORT1 Controller Registers A-13

External Bus Interface Unit Registers A-14

DMA/Memory DMA Control Registers A-15

PPI Registers A-17

Security Registers A-18

Reset and Booting Registers A-18

TWI Registers A-19

UART1 Controller Registers A-20
Ethernet MAC Registers A-20
Handshake MDMA Control Registers A-24
HOST DMA Port Registers A-25
GP Counter Registers A-25
NFC Registers A-26
USB Registers A-27
Processor-Specific Memory Registers A-35
Core Timer Registers A-35

TEST FEATURES

JTAG Standard B-1
Boundary-Scan Architecture B-2
Instruction Register B-4
Public Instructions B-5
EXTEST – Binary Code 00000 ..... B-6
SAMPLE/PRELOAD – Binary Code 10000 ..... B-6
BYPASS – Binary Code 11111 ..... B-6
Boundary-Scan Register B-7

GLOSSARY

INDEX

PREFACE

Thank you for purchasing and developing systems using an enhanced Blackfin® processor from Analog Devices.

Purpose of This Manual

The *ADSP-BF52x Blackfin Processor Hardware Reference* provides architectural information about the ADSP-BF522, ADSP-BF523, ADSP-BF524, ADSP-BF525, ADSP-BF526, and ADSP-BF527 processors. This hardware reference provides architectural information about these processors and the peripherals contained within the ADSP-BF52x Blackfin packages. The architectural descriptions cover functional blocks, buses, and ports, including all features and processes that they support. For programming information, see the *Blackfin Processor Programming Reference.* For timing, electrical, and package specifications, see the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet*.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. This manual assumes that the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts (such as the appropriate instruction set reference manuals and data sheets) that describe your target architecture.

What's New in This Manual

This is Revision 1.0 of the ADSP-BF52x Blackfin Processor Hardware Reference. Peripheral chapters have been expanded and reorganized, and modifications and corrections based on errata reports against this manual have been made.

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at
  `http://www.analog.com/processors/technical_support`
- E-mail tools questions to
  `processor.tools.support@analog.com`
- E-mail processor questions to
  `processor.support@analog.com (World wide support)`
  `processor.europe@analog.com (Europe support)`
  `processor.china@analog.com (China support)`
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:
  `Analog Devices, Inc.`
  `One Technology Way`
  `P.O. Box 9106`
  `Norwood, MA 02062-9106`
  `USA`

Product Information

Product information can be obtained from the Analog Devices Web site, VisualDSP++ online Help system, and a technical library CD.

Analog Devices Web Site

The Analog Devices Web site, www.analog.com, provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

To access a complete technical library for each processor family, go to http://www.analog.com/processors/technical_library. The manuals selection opens a list of current manuals related to the product as well as a link to the previous revisions of the manuals. When locating your manual title, note a possible errata check mark next to the title that leads to the current correction report against the manual.

Also note, MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information about products you are interested in. You can choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Visit MyAnalog.com to sign up. If you are a registered user, just log on. Your user name is your e-mail address.

VisualDSP++ Online Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and FLEXnet License Tools software documenta-

tion. You can search easily across the entire VisualDSP++ documentation set for any topic of interest.

For easy printing, supplementary Portable Documentation Format (.pdf) files for all manuals are provided on the VisualDSP++ installation CD.

Each documentation file type is described as follows.

File	Description
.chm	Help system files and manuals in Microsoft help format
.htm or .html	Dinkum Abridged C++ library and FLEXnet License Tools software documentation. Viewing and printing the .html files requires a browser, such as Internet Explorer 6.0 (or higher).
.pdf	VisualDSP++ and processor manuals in PDF format. Viewing and printing the .pdf files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

Technical Library CD

The technical library CD contains seminar materials, product highlights, a selection guide, and documentation files of processor manuals, VisualDSP++ software manuals, and hardware tools manuals for the following processor families: Blackfin, SHARC, TigerSHARC, ADSP-218x, and ADSP-219x.

To order the technical library CD, go to http://www.analog.com/processors/technical_library, navigate to the manuals page for your processor, click the request CD check mark, and fill out the order form.

Data sheets, which can be downloaded from the Analog Devices Web site, change rapidly, and therefore are not included on the technical library CD. Technical manuals change periodically. Check the Web site for the latest manual revisions and associated documentation errata.

EngineerZone

EngineerZone is a technical support forum from Analog Devices. It allows you direct access to ADI technical support engineers. You can search FAQs and technical information to get quick answers to your embedded processing and DSP design questions.

Use EngineerZone to connect with other DSP developers who face similar design challenges. You can also use this open forum to share knowledge and collaborate with the ADI support team and your peers. Visit `http://ez.analog.com` to sign up.

Social Networking Web Sites

You can now follow Analog Devices processor development on Twitter and LinkedIn. To access:

- Twitter: `http://twitter.com/ADISHARC` and `http://twitter.com/blackfin`
- LinkedIn: Network with the LinkedIn group, Analog Devices SHARC or Analog Devices Blackfin: `http://www.linkedin.com`

Notation Conventions

Text conventions used in this manual are identified and described as follows. Additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.

Example	Description
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
	Note: For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	Caution: Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	Warning: Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for the devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

1 INTRODUCTION

The ADSP-BF52x processors are members of the Blackfin processor family that offer significant high performance and low power while retaining their ease-of-use benefits. All parts within the family are pin-compatible, but only the ADSP-BF526 and ADSP-BF527 include an embedded Ethernet MAC module.

Manual Contents

This manual consists of one volume.

- Chapter 1, "Introduction"
  Provides a high level overview of the processor, including peripherals, power management, and development tools.
- Chapter 2, "Chip Bus Hierarchy"
  Describes on-chip buses, including how data moves through the system.
- Chapter 3, "Memory"
  Describes processor-specific memory topics, including L1 memories and processor-specific memory MMRs.
- Chapter 4, "One-Time Programmable Memory"
  Describes the on-chip, one-time-programmable memory array which provides 64k-bits of non-volatile memory for developers to store both public and private data on-chip.

- Chapter 5, “System Interrupts”
  Describes the system peripheral interrupts, including setup and clearing of interrupt requests.
- Chapter 6, “Direct Memory Access”
  Describes the peripheral DMA and Memory DMA controllers. Includes performance, software management of DMA, and DMA errors.
- Chapter 7, “External Bus Interface Unit”
  Describes the external bus interface unit of the processor. The chapter also discusses the asynchronous memory interface, the SDRAM controller (SDC), related registers, and SDC configuration and commands.
- Chapter 8, “Host DMA Port”
  Describes the Host DMA port of the processor. The Host DMA Port (HOSTDP) allows an external host device to be the DMA master to transfer data to and from the Blackfin device. The host device masters the transactions and the Blackfin is a DMA slave device.
- Chapter 9, “General-Purpose Ports”
  Describes the general-purpose I/O ports, including the structure of each port, multiplexing, configuring the pins, and generating interrupts.
- Chapter 10, “General-Purpose Timers”
  Describes the general-purpose timers.
- Chapter 11, “Core Timer”
  Describes the core timer.
- Chapter 12, “Watchdog Timer”
  Describes the watchdog timer.

- Chapter 13, “General-Purpose Counter”
  Describes the general purpose up/down counter which provides support for manually controlled rotary controllers, such as the volume wheel on a radio device. This unit also supports industrial or motor-control type of wheels.
- Chapter 14, “Real-Time Clock”
  The RTC provides a set of digital watch features to the processor, including time of day, alarm, and stopwatch countdown. It is typically used to implement either a real-time watch or a life counter, which counts the elapsed time since the last system reset.
- Chapter 16, “Security”
  Describes the Lockbox™ Secure Technology for Analog Devices Blackfin processors. This comprises a mix of hardware and software mechanisms designed to prevent unauthorized accesses and allow trusted code to execute on the processor.
- Chapter 17, “System Reset and Booting”
  Describes the booting methods, booting process and specific boot modes for the processor.
- Chapter 18, “Dynamic Power Management”
  Describes the clocking, including the PLL, and the dynamic power management controller.
- Chapter 19, “System Design”
  Describes how to use the processor as part of an overall system. It includes information about bus timing and latency numbers, semaphores, and a discussion of the treatment of unused pins.
- Chapter 21, “Ethernet MAC”
  Describes the Ethernet Media Access Controller (MAC) peripheral which provides a 10/100M bit/s Ethernet interface, compliant to IEEE Std. 802.3-2002, between an MII (Media Independent Interface) and the Blackfin peripheral subsystem.

- Chapter 20, “NAND Flash Controller”
  Describes the NAND Flash Controller (NFC)—which is part of the External Bus Interface—of the processor. NAND Flash devices provide high-density, low-cost memory.
- Chapter 15, “Parallel Peripheral Interface”
  Describes the Parallel Peripheral Interface (PPI) of the processor. The PPI is a half-duplex, bidirectional port accommodating up to 16 bits of data and is used for digital video and data converter applications.
- Chapter 22, “SPI-Compatible Port Controller”
  Describes the Serial Peripheral Interface (SPI) port that provides an I/O interface to a variety of SPI compatible peripheral devices.
- Chapter 23, “Two Wire Interface Controller”
  Describes the Two Wire Interface (TWI) controller, which allows a device to interface to an Inter IC bus as specified by the *Philips I^2C Bus Specification version 2.1* dated January 2000.
- Chapter 24, “SPORT Controller”
  Describes the independent, synchronous Serial Port Controller which provides an I/O interface to a variety of serial peripheral devices.
- Chapter 25, “UART Port Controllers”
  Describes the Universal Asynchronous Receiver/Transmitter port that converts data between serial and parallel formats. The UART supports the half-duplex IrDA® SIR protocol as a mode-enabled feature.
- Chapter 26, “USB OTG Controller”
  Describes the USB OTG interface of the processor. This interface provides a low-cost connectivity solution for consumer mobile devices such as cell phones, digital still cameras and MP3 players, allowing these devices to transfer data via a point-to-point USB connection without the need for a PC host.

- Appendix A, “System MMR Assignments”
  Lists the memory-mapped registers included in this manual, their addresses, and cross-references to text.
- Appendix B, “Test Features”
  Describes test features for the processor, discusses the JTAG standard, boundary-scan architecture, instruction and boundary registers, and public instructions.
- Appendix G, “Glossary”
  Contains definitions of terms used in this book, including acronyms.

This hardware reference is a companion document to the Blackfin Processor Programming Reference.

Peripherals

The processor system peripherals include:

- Two memory-to-memory DMAs with handshake DMA
- Event handler with 54 interrupt inputs
- 12 peripheral DMAs (2 mastered by the Ethernet MAC on ADSP-BF527 processors)
- Host DMA port (HOSTDP)
- 48 General-Purpose I/Os (GPIOs)
- Eight 32-bit timer/counters with PWM support
- 32-bit core timer
- Real-Time Clock (RTC) and watchdog timer
- Rotary counter

- Lockbox™ Secure Technology
- OTP Memory
- On-chip PLL capable of 0.5× to 64× frequency multiplication
- Debug/JTAG interface
- IEEE 802.3-compliant 10/100 Ethernet MAC (only on the ADSP-BF527)
- NAND flash controller
- Parallel Peripheral Interface (PPI), supporting ITU-R 656 video data formats
- Serial Peripheral Interface (SPI)-compatible port
- Two-Wire Interface (TWI) controller
- Two dual-channel, full-duplex synchronous Serial Ports (SPORTs), supporting eight stereo I^2S channels
- Two UARTs with IrDA® support
- USB 2.0 high-speed on-the-go (OTG) interface with integrated PHY

These peripherals are connected to the core via several high bandwidth buses, as shown in Figure 1-1.

All of the peripherals, except for general-purpose I/O, TWI, RTC, and timers, are supported by a flexible DMA structure. There are also two separate memory DMA channels dedicated to data transfers between the processor's memory spaces, which include external SDRAM and asynchronous memory. Multiple on-chip buses provide enough bandwidth to keep

the processor core running even when there is also activity on all of the on-chip and external peripherals.

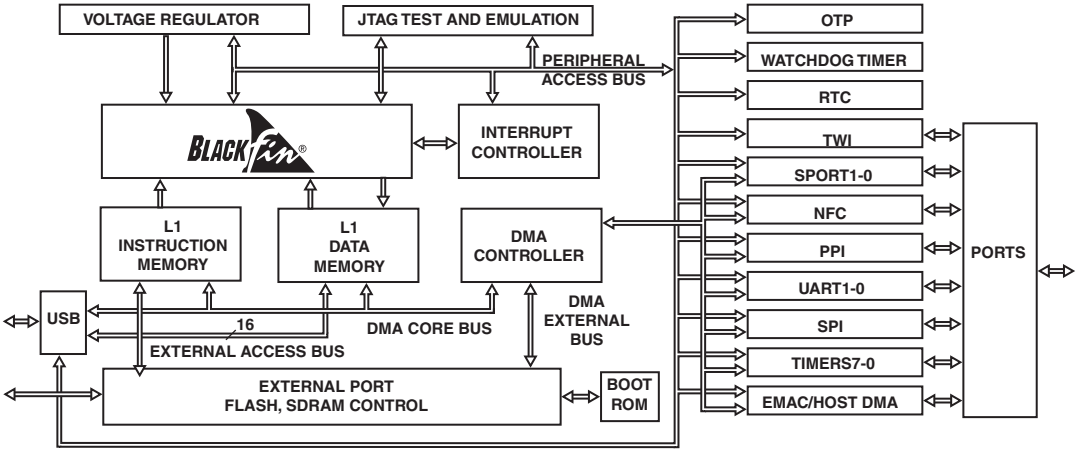


Figure 1-1. ADSP-BF52x Processor Block Diagram

Memory Architecture

The Blackfin processor architecture structures memory as a single, unified 4G byte address space using 32-bit addresses. All resources, including internal memory, external memory, and I/O control registers, occupy separate sections of this common address space. The memory portions of this address space are arranged in a hierarchical structure to provide a good cost/performance balance of some very fast, low latency on-chip memory as cache or SRAM, and larger, lower cost and lower performance off-chip

memory systems. Table 1-1 shows the memory for the ADSP-BF52x processors.

Table 1-1. Memory Configurations

Type of Memory	ADSP-BF52x
Instruction SRAM/cache, lockable by way or line	16K byte
Instruction SRAM	48K byte
Data SRAM/cache	32K byte
Data SRAM	32K byte
Data scratchpad SRAM	4K byte
L3 Boot ROM	32K byte
Total	**164K byte**

The L1 memory system is the primary highest performance memory available to the core. The off-chip memory system, accessed through the external bus interface unit (EBIU), provides expansion with SDRAM, flash memory, and SRAM, optionally accessing up to 132M bytes of physical memory.

The memory DMA controller provides high bandwidth data movement capability. It can perform block transfers of code or data between the internal memory and the external memory spaces.

Internal Memory

The processor has three blocks of on-chip memory that provide high bandwidth access to the core:

- L1 instruction memory, consisting of SRAM and a 4-way set-associative cache. This memory is accessed at full processor speed.
- L1 data memory, consisting of SRAM and/or a 2-way set-associative cache. This memory block is accessed at full processor speed.
- L1 scratchpad RAM, which runs at the same speed as the L1 memories but is only accessible as data SRAM and cannot be configured as cache memory.

External Memory

External (off-chip) memory is accessed via the external bus interface unit (EBIU). This 16-bit interface provides a glueless connection to a bank of synchronous DRAM (SDRAM) and as many as four banks of asynchronous memory devices including flash memory, EPROM, ROM, SRAM, and memory-mapped I/O devices.

The SDRAM controller can be programmed to interface to up to 128M bytes of SDRAM.

The asynchronous memory controller can be programmed to control up to four banks of devices. Each bank occupies a 1M byte segment regardless of the size of the devices used, so that these banks are only contiguous if each is fully populated with 1M byte of memory.

I/O Memory Space

Blackfin processors do not define a separate I/O space. All resources are mapped through the flat 32-bit address space. Control registers for on-chip I/O devices are mapped into memory-mapped registers (MMRs)

at addresses near the top of the 4G byte address space. These are separated into two smaller blocks: one contains the control MMRs for all core functions and the other contains the registers needed for setup and control of the on-chip peripherals outside of the core. The MMRs are accessible only in supervisor mode. They appear as reserved space to on-chip peripherals.

One-Time-Programmable (OTP) Memory

ADSP-BF52x processors also include an on-chip OTP memory array which provides 64K bits of non-volatile memory that can be programmed by the developer one time only. It includes the array and logic to support read access and programming. A mechanism for error correction is provided. Additionally, its pages can be write protected.

The OTP is not part of the Blackfin linear memory map. OTP memory is not accessed directly using the Blackfin memory map; rather, it is accessed via four 32-bit-wide registers (`OTP_DATA3-0`) that act as the OTP memory read/write buffer.

This memory is organized into 512 pages, each comprised of 128 bits and equally separated into two distinct areas with privileged access dependant upon modes of operation when security features are utilized. Approximately 400 pages are available for developer use. The remaining 100 pages are utilized for page protection bits, error correction, and Analog Devices factory-reserved areas. One area is read/write accessible at all time (Public OTP Memory). The second area maintains privileged access and can only be accessed (read/write) upon entry to Secure Mode when security features are utilized (Private OTP Memory).

All together, OTP memory provides a means to store Public Keys in Public OTP Memory or secrets such as Private Keys or Symmetric Keys in Private OTP Memory. One page of the Public OTP Memory is initialized in the Analog Devices factory with a Unique Chip ID.

This OTP memory provides a means to store public and private cipher keys as well as chip, customer, and factory identification data.

DMA Support

The processor has a DMA controller which supports automated data transfers with minimal overhead for the core. DMA transfers can occur between the internal memories and any of its DMA-capable peripherals. Additionally, DMA transfers can be accomplished between any of the DMA-capable peripherals and external devices connected to the external memory interfaces, including the SDRAM controller and the asynchronous memory controller. DMA-capable peripherals include the SPORTs, SPI ports, UARTs, and PPI. For the ADSP-BF527 processor, Ethernet is also a DMA-capable peripheral. Each individual DMA-capable peripheral has at least one dedicated DMA channel.

The DMA controller supports both one-dimensional (1D) and two-dimensional (2-D) DMA transfers. DMA transfer initialization can be implemented from registers or from sets of parameters called descriptor blocks.

The 2-D DMA capability supports arbitrary row and column sizes up to 64K elements by 64K elements, and arbitrary row and column step sizes up to +/- 32K elements. Furthermore, the column step size can be less than the row step size, allowing implementation of interleaved data-streams. This feature is especially useful in video applications where data can be de-interleaved on the fly.

Examples of DMA types supported include:

- A single, linear buffer that stops upon completion
- A circular, auto-refreshing buffer that interrupts on each full or fractionally full buffer
- 1-D or 2-D DMA using a linked list of descriptors
- 2-D DMA using an array of descriptors specifying only the base DMA address within a common page

In addition to the dedicated peripheral DMA channels, there are two separate pairs of memory DMA channels provided for transfers between the various memories of the system. This enables transfers of blocks of data between any of the memories—including external SDRAM, ROM, SRAM, and flash memory—with minimal processor intervention. Memory DMA transfers can be controlled by a very flexible descriptor-based methodology or by a standard register-based autobuffer mechanism.

The ADSP-BF52x processors also include a handshake DMA capability via dual external DMA request pins when used in conjunction with the external bus interface unit (EBIU). This functionality can be used when a high speed interface is required for external FIFOs and high bandwidth communications peripherals such as USB 2.0. It allows control of the number of data transfers for MDMA. The number of transfers per edge is programmable. This feature can be programmed to allow MDMA to have an increased priority on the external bus relative to the core.

External Bus Interface Unit

The external bus interface unit (EBIU) on the processor interfaces with a wide variety of industry-standard memory devices. The controller consists of an SDRAM controller and an asynchronous memory controller.

SDRAM Controller

The SDRAM controller provides an interface to a single bank of industry-standard SDRAM devices or DIMMs. The bank can be configured to contain between 16M and 128M bytes of memory.

A set of programmable timing parameters is available to configure the SDRAM bank to support slower memory devices. The memory bank is 16 bits wide for minimum device count and lower system cost.

Asynchronous Controller

The asynchronous memory controller provides a configurable interface for up to four separate banks of memory or I/O devices. Each bank can be independently programmed with different timing parameters. This allows connection to a wide variety of memory devices, including SRAM, ROM, and flash EPROM, as well as I/O devices that interface with standard memory control lines. Each bank occupies a 1M byte window in the processor address space, but if not fully populated, these are not made contiguous by the memory controller. The banks are 16 bits wide, for interfacing to a range of memories and I/O devices.

Ports

Because of the rich set of peripherals, the ADSP-BF52x processor groups the many peripheral signals to four ports—port F, port G, port H, and port J. Most of the associated pins are shared by multiple signals. The ports function as multiplexer controls. The ports have programmable hysteresis.

General-Purpose I/O (GPIO)

The ADSP-BF52x processors have 48 bi-directional, general-purpose I/O (GPIO) pins allocated across three separate GPIO modules—PORTFIO, PORTGIO, and PORTHIO, associated with port F, port G, and port H, respectively. Port J does not provide GPIO functionality. Each GPIO-capable pin shares functionality with other ADSP-BF52x processor peripherals via a multiplexing scheme; however, the GPIO functionality is the default state of the device upon powerup. Neither GPIO output or input drivers are active by default. Each general-purpose port pin can be

individually controlled by manipulation of the port control, status, and interrupt registers:

- GPIO direction control register – Specifies the direction of each individual GPIO pin as input or output.
- GPIO control and status registers – The ADSP-BF52x processors employ a "write one to modify" mechanism that allows any combination of individual GPIO pins to be modified in a single instruction, without affecting the level of any other GPIO pins. Four control registers are provided. One register is written in order to set pin values, one register is written in order to clear pin values, one register is written in order to toggle pin values, and one register is written in order to specify a pin value. Reading the GPIO status register allows software to interrogate the sense of the pins.
- GPIO interrupt mask registers – The two GPIO interrupt mask registers allow each individual GPIO pin to function as an interrupt to the processor. Similar to the two GPIO control registers that are used to set and clear individual pin values, one GPIO interrupt mask register sets bits to enable interrupt function, and the other GPIO interrupt mask register clears bits to disable interrupt function. GPIO pins defined as inputs can be configured to generate hardware interrupts, while output pins can be triggered by software interrupts.
- GPIO interrupt sensitivity registers – The two GPIO interrupt sensitivity registers specify whether individual pins are level- or edge-sensitive and specify—if edge-sensitive—whether just the rising edge or both the rising and falling edges of the signal are significant. One register selects the type of sensitivity, and one register selects which edges are significant for edge-sensitivity.

Two-Wire Interface

The Two-Wire Interface (TWI) is fully compatible with the widely used I^2C bus standard. It was designed with a high level of functionality and is compatible with multi-master, multi-slave bus configurations. To preserve processor bandwidth, the TWI controller can be set up and a transfer initiated with interrupts only to service FIFO buffer data reads and writes. Protocol related interrupts are optional.

The TWI externally moves 8-bit data while maintaining compliance with the I^2C bus protocol. The *Philips I^2C Bus Specification version 2.1* covers many variants of I^2C. The TWI controller includes these features:

- Simultaneous master and slave operation on multiple device systems
- Support for multi-master data arbitration
- 7-bit addressing
- 100K bits/second and 400K bit/second data rates
- General call address support
- Master clock synchronization and support for clock low extension
- Separate multiple-byte receive and transmit FIFOs
- Low interrupt rate
- Individual override control of data and clock lines in the event of bus lock-up
- Input filter for spike suppression
- Serial camera control bus support as specified in the *OmniVision Serial Camera Control Bus (SCCB) Functional Specification version 2.1*

Ethernet MAC

The Ethernet Media Access Controller (MAC) peripheral for the ADSP-BF527 processors provides a 10/100M bit/second Ethernet interface, compliant with IEEE Std. 802.3-2002, between a Media Independent Interface (MII) and the Blackfin peripheral subsystem. The MAC operates in both half-duplex and full-duplex modes. It provides programmable enhanced features designed to minimize bus utilization and pre- or post-message processing. The connection to the external physical layer device (PHY) is achieved via the MII or a Reduced Media Independent Interface (RMII). The RMII provides data buses half as wide (2 bit vs. 4 bit) as those of an MII, operating at double the frequency.

The MAC is clocked internally from the `CLKIN` pin on the processor. A buffered version of this clock can also be used to drive the external PHY via the `CLKBUF` pin. A 25 MHz source should be used with an MII PHY. A 50 MHz clock source is required to drive an RMII PHY.

Parallel Peripheral Interface

The processor provides a Parallel Peripheral Interface (PPI) that can connect directly to parallel A/D and D/A converters, ITU-R 601/656 video encoders and decoders, and other general-purpose peripherals. The PPI consists of a dedicated input clock pin and three multiplexed frame sync pins. The input clock supports parallel data rates up to half the system clock rate.

In ITU-R 656 modes, the PPI receives and parses a data stream of 8-bit or 10-bit data elements. On-chip decode of embedded preamble control and synchronization information is supported.

Three distinct ITU-R 656 modes are supported:

- Active video only - The PPI does not read in any data between the End of Active Video (EAV) and Start of Active Video (SAV) preamble symbols, or any data present during the vertical blanking intervals. In this mode, the control byte sequences are not stored to memory; they are filtered by the PPI.
- Vertical blanking only - The PPI only transfers Vertical Blanking Interval (VBI) data, as well as horizontal blanking information and control byte sequences on VBI lines.
- Entire field - The entire incoming bitstream is read in through the PPI. This includes active video, control preamble sequences, and ancillary data that may be embedded in horizontal and vertical blanking intervals.

Though not explicitly supported, ITU-R 656 output functionality can be achieved by setting up the entire frame structure (including active video, blanking, and control information) in memory and streaming the data out the PPI in a frame sync-less mode. The processor's 2-D DMA features facilitate this transfer by allowing the static frame buffer (blanking and control codes) to be placed in memory once, and simply updating the active video information on a per-frame basis.

The general-purpose modes of the PPI are intended to suit a wide variety of data capture and transmission applications. The modes are divided into four main categories, each allowing up to 16 bits of data transfer per `PPI_CLK` cycle:

- Data receive with internally generated frame syncs
- Data receive with externally generated frame syncs
- Data transmit with internally generated frame syncs
- Data transmit with externally generated frame syncs

These modes support ADC/DAC connections, as well as video communication with hardware signalling. Many of the modes support more than one level of frame synchronization. If desired, a programmable delay can be inserted between assertion of a frame sync and reception/transmission of data.

SPORT Controllers

The processor incorporates two dual-channel synchronous serial ports (SPORT0 and SPORT1) for serial and multiprocessor communications. The SPORTs support these features:

- Bidirectional, I^2S capable operation

  Each SPORT has two sets of independent transmit and receive pins, which enable eight channels of I^2S stereo audio.

- Buffered (eight-deep) transmit and receive ports

  Each port has a data register for transferring data words to and from other processor components and shift registers for shifting data in and out of the data registers.

- Clocking

  Each transmit and receive port can either use an external serial clock or can generate its own in a wide range of frequencies.

- Word length

  Each SPORT supports serial data words from 3 to 32 bits in length, transferred in most significant bit first or least significant bit first format.

- Framing

  Each transmit and receive port can run with or without frame sync signals for each data word. Frame sync signals can be generated internally or externally, active high or low, and with either of two pulse widths and early or late frame sync.

- Companding in hardware

  Each SPORT can perform A-law or μ-law companding according to ITU recommendation G.711. Companding can be selected on the transmit and/or receive channel of the SPORT without additional latencies.

- DMA operations with single cycle overhead

  Each SPORT can automatically receive and transmit multiple buffers of memory data. The processor can link or chain sequences of DMA transfers between a SPORT and memory.

- Interrupts

  Each transmit and receive port generates an interrupt upon completing the transfer of a data word or after transferring an entire data buffer or buffers through DMA.

- Multichannel capability

  Each SPORT supports 128 channels out of a 1024-channel window and is compatible with the H.100, H.110, MVIP-90, and HMVIP standards.

Serial Peripheral Interface (SPI) Port

The processor has an SPI-compatible port that enables the processor to communicate with multiple SPI-compatible devices.

The SPI interface uses three pins for transferring data: two data pins and a clock pin. An SPI chip select input pin lets other SPI devices select the processor, and seven SPI chip select output pins let the processor select other SPI devices. The SPI select pins are reconfigured general-purpose I/O pins. Using these pins, the SPI port provides a full-duplex, synchronous serial interface, which supports both master and slave modes and multimaster environments.

The SPI port's baud rate and clock phase/polarities are programmable, and it has an integrated DMA controller, configurable to support either transmit or receive datastreams. The SPI's DMA controller can only service unidirectional accesses at any given time.

During transfers, the SPI port simultaneously transmits and receives by serially shifting data in and out of its two serial data lines. The serial clock line synchronizes the shifting and sampling of data on the two serial data lines.

Timers

There are nine general-purpose programmable timer units in the processor. Eight timers have an external pin that can be configured either as a Pulse Width Modulator (PWM) or timer output, as an input to clock the timer, or as a mechanism for measuring pulse widths of external events. These timer units can be synchronized to an external clock input connected to the `PF1` pin, an external clock input to the `PPI_CLK` pin, or to the internal `SCLK`.

The timer units can be used in conjunction with the UARTs to measure the width of the pulses in the datastream to provide an autobaud detect function for a serial channel.

The timers can generate interrupts to the processor core to provide periodic events for synchronization, either to the processor clock or to a count of external signals.

In addition to the eight general-purpose programmable timers, a 9th timer is also provided. This extra timer is clocked by the internal processor clock and is typically used as a system tick clock for generation of operating system periodic interrupts.

UART Ports

The processor provides two half-duplex Universal Asynchronous Receiver/Transmitter (UART) ports, which are fully compatible with PC-standard UARTs. The UART ports provide a simplified UART interface to other peripherals or hosts, providing half-duplex, DMA-supported, asynchronous transfers of serial data. The UART ports include support for 5 to 8 data bits; 1 or 2 stop bits; and none, even, or odd parity. The UART ports support two modes of operation:

- Programmed I/O

  The processor sends or receives data by writing or reading I/O-mapped UART registers. The data is double buffered on both transmit and receive.

- Direct Memory Access (DMA)

  The DMA controller transfers both transmit and receive data. This reduces the number and frequency of interrupts required to transfer data to and from memory. Each of the two UARTs have two

dedicated DMA channels, one for transmit and one for receive. These DMA channels have lower priority than most DMA channels because of their relatively low service rates.

The UARTs' baud rate, serial data format, error code generation and status, and interrupts can be programmed to support:

- Wide range of bit rates
- Data formats from 7 to 12 bits per frame
- Generation of maskable interrupts to the processor by both transmit and receive operations

In conjunction with the general-purpose timer functions, autobaud detection is supported.

The capabilities of the UART ports are further extended with support for the Infrared Data Association (IrDA®) Serial Infrared Physical Layer Link Specification (SIR) protocol.

Security

ADSP-BF52x processors provides security features (Blackfin Lockbox™ Secure Technology) that enable customer applications to use secure protocols, consisting of code authentication and execution of code within a secure environment. Implementing secure protocols on Blackfin processors involves a combination of hardware and software components.

Together these components protect secure memory spaces and restrict control of security features to authenticated developer code.

- Blackfin Lockbox Secure Technology incorporates a secure hardware platform for *confidentiality* and *integrity* protection of secure code and data with *authenticity* maintained by secure software.
- This secure platform provides:
- A secure execution mode
- Secure storage for on-chip keys
- On-chip secure ROM
- Secure RAM
- Access to code and data in the secure domain is monitored by the hardware and any unauthorized access to the secure domain is prevented.
- The secure ROM code establishes the *root of trust* for the secure software in the system.
- The secure RAM provides *integrity* protection and *confidentiality* for authenticated code and data.
- User-defined cipher key(s) and ID(s) can be securely stored in the on-chip OTP memory.
- Every processor ships from the ADI factory with a unique chip ID value stored in publicly accessible OTP memory area.

Real-Time Clock

The processor's Real-Time Clock (RTC) provides a robust set of digital watch features, including current time, stopwatch, and alarm. The RTC is

clocked by a 32.768 kHz crystal external to the processor. The RTC peripheral has dedicated power supply pins, so that it can remain powered up and clocked even when the rest of the processor is in a low power state. The RTC provides several programmable interrupt options, including interrupt per second, minute, hour, or day clock ticks, interrupt on programmable stopwatch countdown, or interrupt at a programmed alarm time.

The 32.768 kHz input clock frequency is divided down to a 1 Hz signal by a prescaler. The counter function of the timer consists of four counters: a 60 second counter, a 60 minute counter, a 24 hours counter, and a 32768 day counter.

When enabled, the alarm function generates an interrupt when the output of the timer matches the programmed value in the alarm control register. There are two alarms. The first alarm is for a time of day. The second alarm is for a day and time of that day.

The stopwatch function counts down from a programmed value, with one minute resolution. When the stopwatch is enabled and the counter underflows, an interrupt is generated.

Like the other peripherals, the RTC can wake up the processor from sleep mode or deep sleep mode upon generation of any RTC wakeup event. An RTC wakeup event can also wake up the on-chip internal voltage regulator from a powered down state.

Watchdog Timer

The processor includes a 32-bit timer that can be used to implement a software watchdog function. A software watchdog can improve system availability by forcing the processor to a known state through generation of a hardware reset, nonmaskable interrupt (NMI), or general-purpose interrupt, if the timer expires before being reset by software. The programmer initializes the count value of the timer, enables the appropriate

interrupt, then enables the timer. Thereafter, the software must reload the counter before it counts to zero from the programmed value. This protects the system from remaining in an unknown state where software that would normally reset the timer has stopped running due to an external noise condition or software error.

If configured to generate a hardware reset, the watchdog timer resets both the CPU and the peripherals. After a reset, software can determine if the watchdog was the source of the hardware reset by interrogating a status bit in the watchdog control register.

The timer is clocked by the system clock (SCLK), at a maximum frequency of f_{SCLK}.

Clock Signals

The processor can be clocked by an external crystal, a sine wave input, or a buffered, shaped clock derived from an external clock oscillator.

This external clock connects to the processor CLKIN pin. The CLKIN input cannot be halted, changed, or operated below the specified frequency during normal operation. This clock signal should be a TTL-compatible signal.

The core clock (CCLK) and system peripheral clock (SCLK) are derived from the input clock (CLKIN) signal. An on-chip Phase Locked Loop (PLL) is capable of multiplying the CLKIN signal by a user-programmable (0.5× to 64×) multiplication factor (bounded by specified minimum and maximum VCO frequencies). The default multiplier is 10×, but it can be modified by a software instruction sequence. On-the-fly frequency changes can be made by simply writing to the PLL_DIV register.

All on-chip peripherals are clocked by the system clock (SCLK). The system clock frequency is programmable by means of the SSEL[3:0] bits of the PLL_DIV register.

Dynamic Power Management

The processor provides four operating modes, each with a different performance/power profile. In addition, dynamic power management provides the control functions to dynamically alter the processor core supply voltage to further reduce power dissipation. Control of clocking to each of the peripherals also reduces power consumption.

Full-On Mode (Maximum Performance)

In the full-on mode, the PLL is enabled, not bypassed, providing the maximum operational frequency. This is the normal execution state in which maximum performance can be achieved. The processor core and all enabled peripherals run at full speed.

Active Mode (Moderate Power Savings)

In the active mode, the PLL is enabled, but bypassed. Because the PLL is bypassed, the processor's core clock (CCLK) and system clock (SCLK) run at the input clock (CLKIN) frequency. In this mode, the CLKIN to VCO multiplier ratio can be changed, although the changes are not realized until the full on mode is entered. DMA access is available to appropriately configured L1 memories.

In the active mode, it is possible to disable the PLL through the PLL control register (PLL_CTL). If disabled, the PLL must be re-enabled before transitioning to the full on or sleep modes.

Sleep Mode (High Power Savings)

The sleep mode reduces power dissipation by disabling the clock to the processor core (CCLK). The PLL and system clock (SCLK), however, continue to operate in this mode. Typically an external event or RTC activity will wake up the processor. When in the sleep mode, assertion of any

interrupt causes the processor to sense the value of the bypass bit (BYPASS) in the PLL control register (PLL_CTL). If bypass is disabled, the processor transitions to the full on mode. If bypass is enabled, the processor transitions to the active mode.

When in the sleep mode, system DMA access to L1 memory is not supported.

Deep Sleep Mode (Maximum Power Savings)

The deep sleep mode maximizes dynamic power savings by disabling the processor core and synchronous system clocks (CCLK and SCLK). Asynchronous systems, such as the RTC, may still be running, but cannot access internal resources or external memory. This powered-down mode can only be exited by assertion of the reset interrupt or by an asynchronous interrupt generated by the RTC. When in deep sleep mode, an RTC asynchronous interrupt causes the processor to transition to the active mode. Assertion of $\overline{\text{RESET}}$ while in deep sleep mode causes the processor to transition to the full on mode.

Hibernate State

For lowest possible power dissipation, this state allows the internal supply (V_{DDINT}) to be powered down, while keeping the I/O supply (V_{DDEXT}) running. Although not strictly an operating mode like the four modes detailed above, it is illustrative to view it as such.

Voltage Regulation

The ADSP-BF523, ADSP-BF525, ADSP-BF527 processors provide an on-chip voltage regulator that can generate V_{DDINT} from an external supply. Figure 18-3 on page 18-18 shows the typical external components required to complete the power management system. The regulator con-

trols the internal logic voltage levels and is programmable with the voltage regulator control register (VR_CTL) in increments of 50 mV. To reduce standby power consumption, the internal voltage regulator can be programmed to remove power to the processor core while keeping I/O power supplied. While in this state, V_{DDEXT} can still be applied, eliminating the need for external buffers. The regulator can also be disabled and bypassed at the user's discretion.

Instruction Set Description

The Blackfin processor family assembly language instruction set employs an algebraic syntax designed for ease of coding and readability. Refer to the Blackfin Processor Programming Reference for detailed information. The instructions have been specifically tuned to provide a flexible, densely encoded instruction set that compiles to a very small final memory size. The instruction set also provides fully featured multifunction instructions that allow the programmer to use many of the processor core resources in a single instruction. Coupled with many features more often seen on microcontrollers, this instruction set is very efficient when compiling C and C++ source code. In addition, the architecture supports both user (algorithm/application code) and supervisor (O/S kernel, device drivers, debuggers, ISRs) modes of operation, allowing multiple levels of access to core resources.

The assembly language, which takes advantage of the processor's unique architecture, offers these advantages:

- Embedded 16/32-bit microcontroller features, such as arbitrary bit and bit field manipulation, insertion, and extraction; integer operations on 8-, 16-, and 32-bit data types; and separate user and supervisor stack pointers
- Seamlessly integrated DSP/CPU features optimized for both 8-bit and 16-bit operations

- A multi-issue load/store modified Harvard architecture, which supports two 16-bit MAC or four 8-bit ALU + two load/store + two pointer updates per cycle
- All registers, I/O, and memory mapped into a unified 4G byte memory space, providing a simplified programming model

Code density enhancements include intermixing of 16- and 32-bit instructions with no mode switching or code segregation. Frequently used instructions are encoded in 16 bits.

Development Tools

The processor is supported with a complete set of CROSSCORE® software and hardware development tools, including Analog Devices emulators and the VisualDSP++® development environment. The same emulator hardware that supports other Analog Devices products also fully emulates the Blackfin processor family.

The VisualDSP++ project management environment lets programmers develop and debug an application. This environment includes an easy-to-use assembler that is based on an algebraic syntax, an archiver (librarian/library builder), a linker, a loader, a cycle-accurate instruction-level simulator, a C/C++ compiler, and a C/C++ runtime library that includes DSP and mathematical functions. A key point for these tools is C/C++ code efficiency. The compiler has been developed for efficient translation of C/C++ code to Blackfin processor assembly. The Blackfin processor has architectural features that improve the efficiency of compiled C/C++ code.

Debugging both C/C++ and assembly programs with the VisualDSP++ debugger, programmers can:

- View mixed C/C++ and assembly code (interleaved source and object information)
- Insert breakpoints
- Set conditional breakpoints on registers, memory, and stacks
- Trace instruction execution
- Perform linear or statistical profiling of program execution
- Fill, dump, and graphically plot the contents of memory
- Perform source level debugging
- Create custom debugger windows

The VisualDSP++ Integrated Development and Debugging Environment (IDDE) lets programmers define and manage software development. Its dialog boxes and property pages let programmers configure and manage all development tools, including color syntax highlighting in the VisualDSP++ editor. These capabilities permit programmers to:

- Control how the development tools process inputs and generate outputs
- Maintain a one-to-one correspondence with the tool's command-line switches

The VisualDSP++ Kernel (VDK) incorporates scheduling and resource management tailored specifically to address the memory and timing constraints of DSP programming. These capabilities enable engineers to develop code more effectively, eliminating the need to start from the very beginning, when developing new application code. The VDK features include threads, critical and unscheduled regions, semaphores, events, and device flags. The VDK also supports priority-based, pre-emptive, coopera-

tive and time-sliced scheduling approaches. In addition, the VDK was designed to be scalable. If the application does not use a specific feature, the support code for that feature is excluded from the target system.

Because the VDK is a library, a developer can decide whether to use it or not. The VDK is integrated into the VisualDSP++ development environment but can also be used with standard command-line tools. The VDK development environment assists in managing system resources, automating the generation of various VDK-based objects, and visualizing the system state during application debug.

Analog Devices emulators use the IEEE 1149.1 JTAG test access port of the processor to monitor and control the target board processor during emulation. The emulator provides full speed emulation, allowing inspection and modification of memory, registers, and processor stacks. Nonintrusive in-circuit emulation is assured by the use of the processor's JTAG interface—the emulator does not affect target system loading or timing.

In addition to the software and hardware development tools available from Analog Devices, third parties provide a wide range of tools supporting the Blackfin processor family. Hardware tools include the ADSP-BF52x EZ-KIT Lite standalone evaluation/development cards. Third party software tools include DSP libraries, real-time operating systems, and block diagram design tools.

2 CHIP BUS HIERARCHY

This chapter discusses on-chip buses, how data moves through the system, and other factors that determine the system organization. Following an overview and a list of key features is a block diagram of the chip bus hierarchy and a description of its operation. The chapter concludes with details about the system interconnects and associated system buses.

Overview

The ADSP-BF52x Blackfin processors feature a powerful chip bus hierarchy on which all data movement between the processor core, internal memory, external memory, and its rich set of peripherals occurs. The chip bus hierarchy includes the controllers for system interrupts, test/emulation, and clock and power management. Synchronous clock domain conversion is provided to support clock domain transactions between the core and the system.

The processor system includes:

- The peripheral set (timers, real-time clock, TWI, Ethernet MAC (ADSP-BF527), USB 2.0, GPIOs, UARTs, SPORTs, PPI, watchdog timer, and SPI)
- The external bus interface unit (EBIU)
- The host DMA port (HOSTDP)

- The Direct Memory Access (DMA) controller
- The interfaces between these, the system, and the optional external (off-chip) resources

The following sections describe the on-chip interfaces between the system and the peripherals via the:

- Peripheral Access Bus (PAB)
- DMA Access Bus (DAB)
- DMA Core Bus (DCB)
- DMA External Bus (DEB)
- External Access Bus (EAB)

The external bus interface unit (EBIU) is the primary chip pin bus and is discussed in Chapter 7, "External Bus Interface Unit".

Interface Overview

Figure 2-1 shows the core processor and system boundaries as well as the interfaces between them.

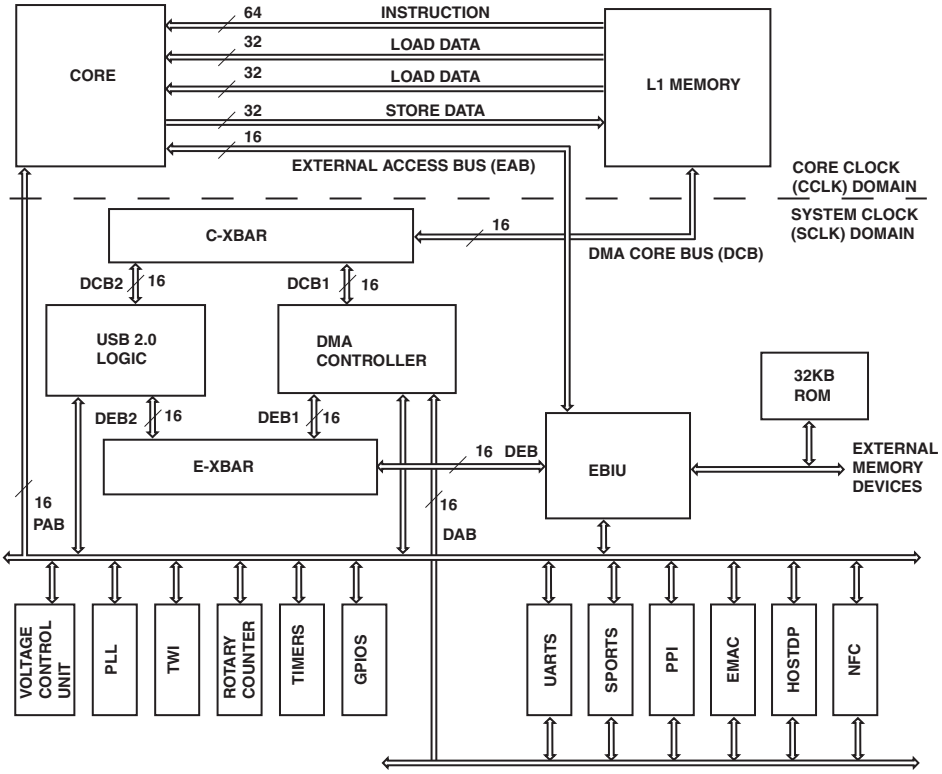


Figure 2-1. Processor Bus Hierarchy

Internal Clocks

The core processor clock (CCLK) rate is highly programmable with respect to CLKIN. The CCLK rate is divided down from the Phase Locked Loop

(PLL) output rate. This divider ratio is set using the CSEL parameter of the PLL divide register.

The PAB, the DAB, the EAB, the DCB, the DEB, the EPB, and the EBIU run at system clock frequency (SCLK domain). This divider ratio is set using the SSEL parameter of the PLL divide register and must be set so that these buses run as specified in the processor data sheet, and slower than or equal to the core clock frequency.

These buses can also be cycled at a programmable frequency to reduce power consumption, or to allow the core processor to run at an optimal frequency. Note all synchronous peripherals derive their timing from the SCLK. For example, the UART clock rate is determined by further dividing this clock frequency.

Core Bus Overview

For the purposes of this discussion, level 1 memories (L1) are included in the description of the core; they have full bandwidth access from the processor core with a 64-bit instruction bus and two 32-bit data buses.

Figure 2-2 shows the core processor and its interfaces to the peripherals and external memory resources.

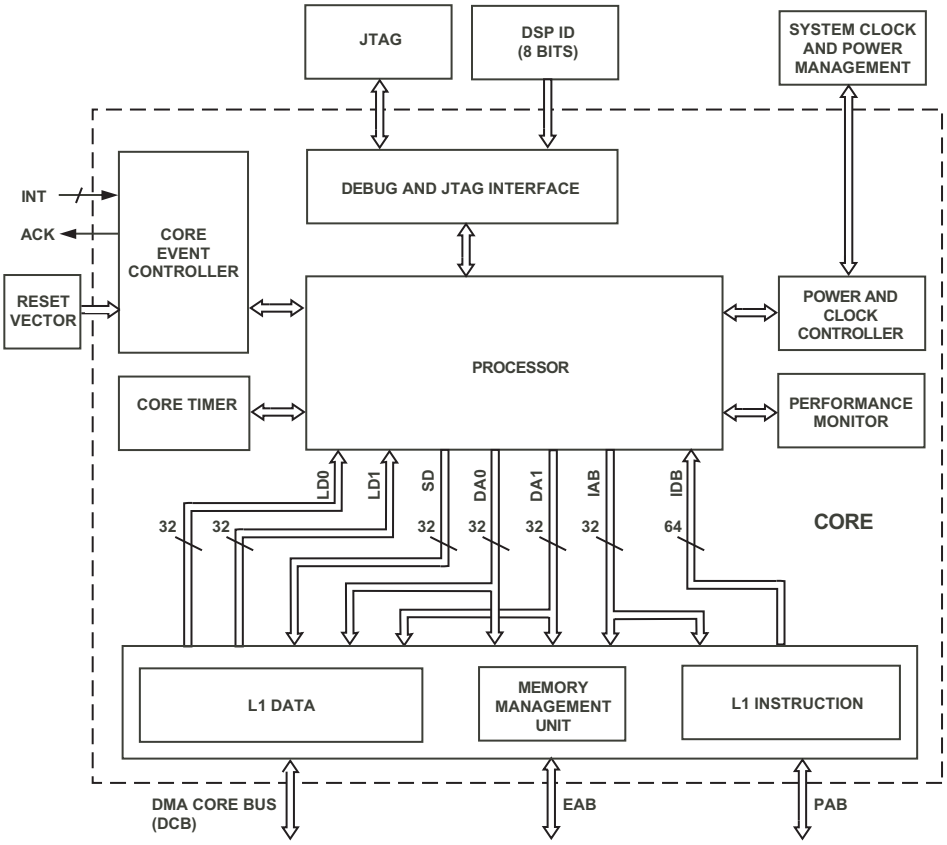


Figure 2-2. Core Block Diagram

The core can generate up to three simultaneous off-core accesses per cycle.

The core bus structure between the processor and L1 memory runs at the full core frequency and has data paths up to 64 bits.

When the instruction request is filled, the 64-bit read can contain a single 64-bit instruction or any combination of 16-, 32-, or 64-bit (partial) instructions.

When cache is enabled, four 64-bit read requests are issued to support 32-byte line fill burst operations. These requests are pipelined so that each transfer after the first is filled in a single, consecutive cycle.

Peripheral Access Bus (PAB)

The processor has a dedicated low latency peripheral bus that keeps core stalls to a minimum and allows for manageable interrupt latencies to time-critical peripherals. All peripheral resources accessed through the PAB are mapped into the system MMR space of the processor memory map. The core accesses system MMR space through the PAB bus.

The core processor has byte addressability, but the programming model is restricted to only 32-bit (aligned) access to the system MMRs. Byte accesses to this region are not supported.

PAB Arbitration

The core is the only master on this bus. No arbitration is necessary.

PAB Agents (Masters, Slaves)

The processor core can master bus operations on the PAB. All peripherals have a peripheral bus slave interface which allows the core to access control and status state. These registers are mapped into the system MMR space of the memory map. Appendix B lists system MMR addresses.

The slaves on the PAB bus are:

- System event controller
- Clock and power management controller

- Watchdog timer
- Real-time clock (RTC)
- Timer 0–7
- SPORT0–1
- SPI
- Ports
- UART0–1
- PPI
- TWI
- NAND flash controller (NFC)
- Ethernet MAC
- USB 2.0
- Asynchronous memory controller (AMC)
- SDRAM controller (SDC)
- DMA controller

PAB Performance

For the PAB, the primary performance criteria is latency, not throughput. Transfer latencies for both read and write transfers on the PAB are two SCLK cycles.

For example, the core can transfer up to 32 bits per access to the PAB slaves. With the core clock running at 2× the frequency of the system clock, the first and subsequent system MMR read or write accesses take four core clocks (CCLK) of latency.

The PAB has a maximum frequency of SCLK.

DMA Access Bus (DAB), DMA Core Bus (DCB), DMA External Bus (DEB)

The DAB, DCB, and DEB buses provide a means for DMA-capable peripherals to gain access to on-chip and off-chip memory with little or no degradation in core bandwidth to memory.

DAB, DCB, DEB Arbitration

Sixteen DMA channels and bus masters support the DMA-capable peripherals in the processor system. The twelve peripheral DMA channel controllers can transfer data between peripherals and internal or external memory. Both the read and write channels of the dual-stream memory DMA controller access their descriptor lists through the DAB.

The DCB has priority over the core processor on arbitration into L1 configured as SRAM.

The following priority applies to external bus accesses, between core and DMA:

1. Core Locked Access (Testset)
2. Urgent DMA Access
3. Core Access
4. Normal (Regular) DMA Access

The processor has a programmable priority arbitration policy on the DAB. Table 2-1 shows the default arbitration priority.

Table 2-1. DAB, DCB, and DEB Arbitration Priority

DAB, DCB, DEB Master	Default Arbitration Priority
PPI receive/transmit or NFC	0 - highest
Ethernet receive	1
Ethernet transmit or NFC	2
SPORT0 receive	3
SPORT0 transmit	4
SPORT1 receive	5
SPORT1 transmit	6
SPI receive/transmit	7
UART0 receive	8
UART0 transmit	9
UART1 receive	10
UART1 transmit	11
MDMA stream 0 destination	12
MDMA stream 0 source	13
MDMA stream 1 destination	14
MDMA stream 1 source	15 - lowest

DCB Sharing

USB has a separate DMA Controller and supports eight channels. The Peripheral DMA bus (DCB1) and USB DMA bus (DCB2) have to share the same L1 memory bus (DCB). This sharing happens in the C-XBAR. Arbitration logic determines who gets the bus when requests from DCB1 and DCB2 arrive in the same cycle. There are control bits which deter-

mine how the arbitration is to be performed. Table 2-2 on page 2-10 describes the arbitration scheme. In Table 2-2 on page 2-10 the DCBx Urgency columns denote that one or more peripherals on the specified DCBx bus is making an urgent request. For more information, see "Temporary DMA Urgency" on page 6-46.

Table 2-2. Arbitration Scheme for DCB

DCB_ROT_PRIO	DCB1_PRIO_HI	DCB1 Urgency	DCB2 Urgency	Access to DCB (L1 bus)
0	0	0	0	DCB2
0	0	X	1	DCB2
0	0	1	0	DCB1
0	1	0	0	DCB1
0	1	1	X	DCB1
0	1	0	1	DCB2
1	X	0	0	Ping-Pong
1	X	0	1	DCB2
1	X	1	0	DCB1
1	X	1	1	Ping –Pong

1. DCB_ROT_PRIO is configured in the SYSCR register. When this bit is set to "1", a rotating priority scheme is selected between DCB1 and DCB2, and the DCB_PRIO_HI setting is ignored.

   Note that the rotating priority scheme switches the access between DCB1 and DCB2 at every access to the internal L1 memory. This limits bursty transfers to/from L1 memory to the DMA channel, which could limit efficient utilization of the bus bandwidth.

2. DCB1_PRIO_HI is also be configured in the SYSCR register. When this bit is set to "1", DCB1 has a higher priority than DCB2. If this bit is set to "0", DCB2 has a higher priority than DCB1.

For more information on the SYSCR register, see "System Reset Configuration (SYSCR) Register" on page 17-100.

For L3 access, the L3 memory bus (DEB) is shared between the USB bus (DEB2) and the DEB1 bus which is used for all other accesses (peripheral DMA, core accesses, MDMA). This sharing happens in the E-XBAR. The arbitration logic and the scheme is identical to the C-XBAR. Table 2-3 on page 2-12 indicates the arbitration scheme. In Table 2-3, the DEBx Urgency columns denote that one or more peripherals on the specified

DEBx bus is making an urgent request. For more information, see "Temporary DMA Urgency" on page 6-46.

Table 2-3. Arbitration Scheme for DEB

DEB_ROT_PRIO	DEB1_PRIO_HI	DEB1 Urgency	DEB2 Urgency	Access to DEB (L3 bus)
0	0	0	0	DEB2
0	0	X	1	DEB2
0	0	1	0	DEB1
0	1	0	0	DEB1
0	1	1	X	DEB1
0	1	0	1	DEB2
1	X	0	0	Ping-pong
1	X	0	1	DEB2
1	X	1	0	DEB1
1	X	1	1	Ping-pong

1. `DEB_ROT_PRIO_En` is configured in the `SYSCR` register. When this bit is set to "1", a rotating priority scheme is selected between `DEB1` and `DEB2`, and the `DEB1_PRIO_HI` setting is ignored.

   Note that the rotating priority scheme switches the access between DEB1 and DEB2 at every access to the external memory. This limits bursty transfers to/from the external memory to the DMA channel, which could limit efficient utilization of the bus bandwidth.

2. `DEB1_PRIO_HI` can also configured in the `SYSCR` register, When this bit is set to "1", `DEB1` has higher priority than `DEB2`. If this bit is set to "0", `DEB2` has higher priority than `DEB1`.

For more information on the `SYSCR` register, see "System Reset Configuration (SYSCR) Register" on page 17-100.

Using the CDPRIO Bit to Change Priorities

The core and DMA prioritization over the external bus can also be programmed statically by using the CDPRIO bit in the EBIU_AMGCTL register. By setting the CDPRIO bit in the EBIU_AMGCTL register, all DEB (DEB1 and DEB2) transactions to the external bus have priority over core accesses to external memory. Use of this bit is application dependent. For example, if you are polling a peripheral mapped to asynchronous memory with long access times, by default the core will “win” over DMA requests. By setting the CDPRIO bit, the core would be held off until DMA requests were serviced. Use this bit only if "Temporary Urgent DMA" functionality of DEB1 as controlled automatically by the hardware is insufficient to meet system bandwidth requirements.

The "Temporary Urgent DMA" functionality of DEB2 is not controlled automatically by the hardware. DEB2 requests never go urgent automatically and thus can never beat the core for external memory accesses under temporary urgent conditions. The DEB2_URGENT bit in the USB_PLLOSC_CTRL register can be used to statically assign DEB2 requests higher priority than the core. See “USB PLL OSC Control (USB_PLLOSC_CTRL) Registers” on page 26-140.

DAB Bus Agents (Masters)

All peripherals capable of sourcing a DMA access are masters on this bus, as shown in Table 2-1. A single arbiter supports a programmable priority arbitration policy for access to the DAB.

When two or more DMA master channels are actively requesting the DAB, bus utilization is considerably higher due to the DAB’s pipelined design. Bus arbitration cycles are concurrent with the previous DMA access’s data cycles.

DAB, DCB, and DEB Performance

The processor DAB supports data transfers of 16 bits or 32 bits. The data bus has a 16-bit width with a maximum frequency as specified in the processor data sheet.

The DAB has a dedicated port into L1 memory. No stalls occur as long as the core access and the DMA access are not to the same memory bank (4K byte size for L1). If there is a conflict, DMA is the highest priority requester, followed by the core.

Note that a locked transfer by the core processor (for example, execution of a `TESTSET` instruction) effectively disables arbitration for the addressed memory bank or resource until the memory lock is deasserted. DMA controllers cannot perform locked transfers.

DMA access to L1 memory can only be stalled by an access already in progress from another DMA channel. Latencies caused by these stalls are in addition to any arbitration latencies.

The core processor and the DAB must arbitrate for access to external memory through the EBIU. This additional arbitration latency added to the latency required to read off-chip memory devices can significantly degrade DAB throughput, potentially causing peripheral data buffers to underflow or overflow. If you use DMA peripherals other than the memory DMA controller, and you target external memory for DMA accesses, you need to carefully analyze your specific traffic patterns. Make sure that isochronous peripherals targeting internal memory have enough allocated bandwidth and the appropriate maximum arbitration latencies.

External Access Bus (EAB)

The EAB provides a way for the processor core to directly access off-chip memory.

Arbitration of the External Bus

Arbitration for use of external port bus interface resources is required because of possible contention between the potential masters of this bus. A fixed-priority arbitration scheme is used. That is, core accesses via the EAB will be of higher priority than those from the DMA external bus (DEB).

DEB/EAB Performance

The DEB and the EAB support single word accesses of either 8-bit or 16-bit data types. The DEB and the EAB operate at the same frequency as the PAB and the DAB, up to the maximum `SCLK` frequency specified in the processor data sheet.

Memory DMA transfers can result in repeated accesses to the same memory location. Because the memory DMA controller has the potential of simultaneously accessing on-chip and off-chip memory, considerable throughput can be achieved. The throughput rate for an on-chip/off-chip memory access is limited by the slower of the two accesses.

In the case where the transfer is from on-chip to on-chip memory or from off-chip to off-chip memory, the burst accesses cannot occur simultaneously. The transfer rate is then determined by adding each transfer plus an additional cycle between each transfer.

Table 2-4 shows many types of 16-bit memory DMA transfers. In the table, it is assumed that no other DMA activity is conflicting with ongoing operations. The numbers in the table are theoretical values. These values may be higher when they are measured on actual hardware due to a variety of reasons relating to the device that is connected to the EBIU.

For non-DMA accesses (for example, a core access via the EAB), a 32-bit access to SDRAM (of the form `R0 = [P0];` where P0 points to an address in SDRAM) is always more efficient than executing two 16-bit accesses (of the form `R0 = W[P0++];` where P0 points to an address in SDRAM). In

this example, a 32-bit SDRAM read takes 10 SCLK cycles while two 16-bit reads take 9 SCLK cycles each.

Table 2-4. Performance of DMA Access to External Memory

Source	Destination	Approximate SCLKs For n Words (from start of DMA to interrupt at end)
16-bit SDRAM	L1 data memory	n + 14
L1 data memory	16-bit SDRAM	n + 11
16-bit async memory	L1 data memory	xn + 12, where x is the number of wait states + setup/hold SCLK cycles (minimum x = 2)
L1 data memory	16-bit async memory	xn + 9, where x is the number of wait states + setup/hold SCLK cycles (minimum x = 2)
16-bit SDRAM	16-bit SDRAM	10 + (17n/7)
16-bit async memory	16-bit async memory	10 + 2xn, where x is the number of wait states + setup/hold SCLK cycles (minimum x = 2)
L1 data memory	L1 data memory	2n + 12

3 MEMORY

This chapter discusses memory population specific to the ADSP-BF52x processors. Functional memory architecture is described in the Blackfin Processor Programming Reference.

Memory Architecture

Table 3-1 on page 3-1 provides an overview of the ADSP-BF52x processor system memory map. For a detailed discussion of how to use them, see the Blackfin Processor Programming Reference. Note the architecture does not define a separate I/O space. All resources are mapped through the flat 32-bit address space. The memory is byte-addressable.

The upper portion of internal memory space is allocated to the core and system MMRs. Accesses to this area are allowed only when the processor is in supervisor or emulation mode (see the Operating Modes and States chapter of the Blackfin Processor Programming Reference).

Within the external memory map, four banks of asynchronous memory space and one bank of SDRAM memory are available. Each of the asynchronous banks is 1M byte and the SDRAM bank is up to 128M byte.

Table 3-1. ADSP-BF52x Memory Map

Starting Address	Ending Address	Description
0xFFE0 0000	0xFFFF FFFF	Core MMR
0xFFC0 0000	0xFFDF FFFF	System MMR
0xFFB0 1000	0xFFBF FFFF	reserved

Table 3-1. ADSP-BF52x Memory Map

Starting Address	Ending Address	Description
0xFFB0 0000	0xFFB0 0FFF	Scratchpad SRAM
0xFFA1 4000	0xFFAF FFFF	reserved
0xFFA1 0000	0xFFA1 3FFF	Instruction Bank C SRAM/Cache
0xFFA0 C000	0xFFA0 FFFF	reserved
0xFFA0 8000	0xFFA0 BFFF	Instruction Bank B SRAM
0xFFA0 0000	0xFFA0 7FFF	Instruction Bank A SRAM
0xFF90 8000	0xFF9F FFFF	reserved
0xFF90 4000	0xFF90 7FFF	Data Bank B SRAM/Cache
0xFF90 0000	0xFF90 3FFF	Data Bank B SRAM
0xFF80 8000	0xFF8F FFFF	reserved
0xFF80 4000	0xFF80 7FFF	Data Bank A SRAM/Cache
0xFF80 0000	0xFF80 3FFF	Data Bank A SRAM
0xEF00 4000	0xFF7F FFFF	reserved
0xEF00 0000	0xEF00 7FFF	BOOT ROM (32K Byte)
0x2080 0000	0xEEFF FFFF	reserved
0x2040 0000	0x207F FFFF	reserved
0x2030 0000	0x203F FFFF	Async Bank 3
0x2020 0000	0x202F FFFF	Async Bank 2
0x2010 0000	0x201F FFFF	Async Bank 1
0x2000 0000	0x200F FFFF	Async Bank 0
0x0800 0000	0x1FFF FFFF	reserved
0x0000 0000	0x07FF FFFF	SDRAM

L1 Instruction SRAM

The processor core reads the instruction memory through the 64-bit wide instruction fetch bus. All addresses from this bus are 64-bit aligned. Each instruction fetch can return any combination of 16-, 32- or 64-bit instructions (for example, four 16-bit instructions, two 16-bit instructions and one 32-bit instruction, or one 64-bit instruction).

Table 3-2 lists the memory start locations of the L1 instruction memory subbanks.

Table 3-2. L1 Instruction Memory Subbanks

Memory Subbank	Memory Start Location for ADSP-BF52x Processors
0	0xFFA0 0000
1	0xFFA0 1000
2	0xFFA0 2000
3	0xFFA0 3000
4	0xFFA0 4000
5	0xFFA0 5000
6	0xFFA0 6000
7	0xFFA0 7000
8	0xFFA0 8000
9	0xFFA0 9000
10	0xFFA0 A000
11	0xFFA0 B000
12	0xFFA1 0000
13	0xFFA1 1000
14	0xFFA1 2000
15	0xFFA1 3000

L1 Data SRAM

Table 3-3 shows how the subbank organization is mapped into memory.

Table 3-3. L1 Data Memory SRAM Subbank Start Addresses

Memory Bank and Subbank	ADSP-BF52x Processors
Data Bank A, Subbank 0	0xFF80 0000
Data Bank A, Subbank 1	0xFF80 1000
Data Bank A, Subbank 2	0xFF80 2000
Data Bank A, Subbank 3	0xFF80 3000
Data Bank A, Subbank 4	0xFF80 4000
Data Bank A, Subbank 5	0xFF80 5000
Data Bank A, Subbank 6	0xFF80 6000
Data Bank A, Subbank 7	0xFF80 7000
Data Bank B, Subbank 0	0xFF90 0000
Data Bank B, Subbank 1	0xFF90 1000
Data Bank B, Subbank 2	0xFF90 2000
Data Bank B, Subbank 3	0xFF90 3000
Data Bank B, Subbank 4	0xFF90 4000
Data Bank B, Subbank 5	0xFF90 5000
Data Bank B, Subbank 6	0xFF90 6000
Data Bank B, Subbank 7	0xFF90 7000

L1 Data Cache

When data cache is enabled (controlled by bits `DMC[1:0]` in the `DMEM_CONTROL` register), either 16K byte of data bank A or 16K byte of both data bank A and data bank B can be set to serve as cache.

Boot ROM

The lowest 32K byte of internal memory space is occupied by the boot ROM starting from address 0xEF00 0000. This 16-bit boot ROM is not part of the L1 memory module. Read accesses take one SCLK cycle and no wait states are required. The read-only memory can be read by the core as well as by DMA. It can be cached and protected by CPLD blocks like external memory. The boot ROM not only contains boot-strap loader code, it also provides some subfunctions that are user-callable at runtime. For more information, see "System Reset and Booting" on page 17-1.

External Memory

The external memory space is shown in Figure 3-1 on page 3-1. One of the memory regions is dedicated to SDRAM support. The size of the SDRAM bank is programmable and can range in size from 16M byte to 128M byte. The start address of the bank is 0x0000 0000.

Each of the next four banks contains 1M byte and is dedicated to support asynchronous memories. The start address of the asynchronous memory bank is 0x2000 0000.

Processor-Specific MMRs

The complete set of memory-related MMRs is described in the Blackfin Processor Programming Reference. Several MMRs have bit definitions specific to the processors described in this manual. These registers are described in the following sections.

DMEM_CONTROL Register

The data memory control register (DMEM_CONTROL), shown in Figure 3-1, contains control bits for the L1 data memory.

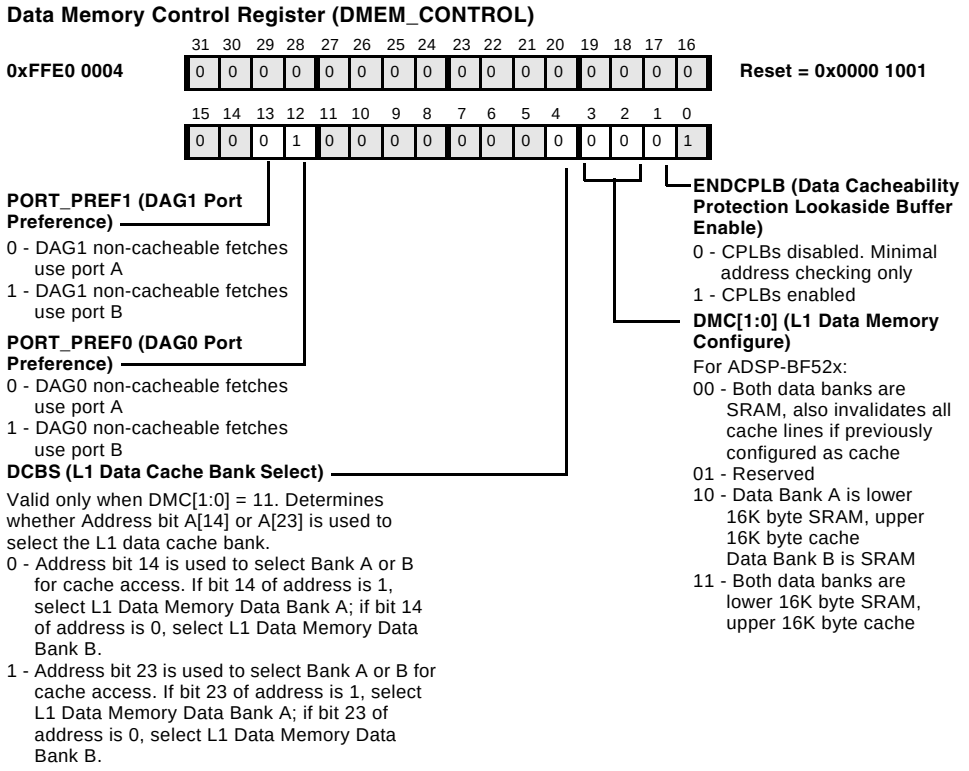

Data Memory Control Register (DMEM_CONTROL)
0xFFE0 0004
Reset = 0x0000 1001
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
PORT_PREF1 (DAG1 Port Preference)
0 - DAG1 non-cacheable fetches use port A
1 - DAG1 non-cacheable fetches use port B
PORT_PREF0 (DAG0 Port Preference)
0 - DAG0 non-cacheable fetches use port A
1 - DAG0 non-cacheable fetches use port B
DCBS (L1 Data Cache Bank Select)
Valid only when DMC[1:0] = 11. Determines whether Address bit A[14] or A[23] is used to select the L1 data cache bank.
0 - Address bit 14 is used to select Bank A or B for cache access. If bit 14 of address is 1, select L1 Data Memory Data Bank A; if bit 14 of address is 0, select L1 Data Memory Data Bank B.
1 - Address bit 23 is used to select Bank A or B for cache access. If bit 23 of address is 1, select L1 Data Memory Data Bank A; if bit 23 of address is 0, select L1 Data Memory Data Bank B.
ENDCPLB (Data Cacheability Protection Lookaside Buffer Enable)
0 - CPLBs disabled. Minimal address checking only
1 - CPLBs enabled
DMC[1:0] (L1 Data Memory Configure)
For ADSP-BF52x:
00 - Both data banks are SRAM, also invalidates all cache lines if previously configured as cache
01 - Reserved
10 - Data Bank A is lower 16K byte SRAM, upper 16K byte cache
Data Bank B is SRAM
11 - Both data banks are lower 16K byte SRAM, upper 16K byte cache

Figure 3-1. L1 Data Memory Control Register

DTEST_COMMAND Register

When the data test command register (DTEST_COMMAND) is written to, the L1 cache data or tag arrays are accessed, and the data is transferred

through the data test data registers (DTEST DATA[1:0]). This register is shown in Figure 3-2.

The data/instruction access bit allows direct access via the DTEST_COMMAND MMR to L1 instruction SRAM.

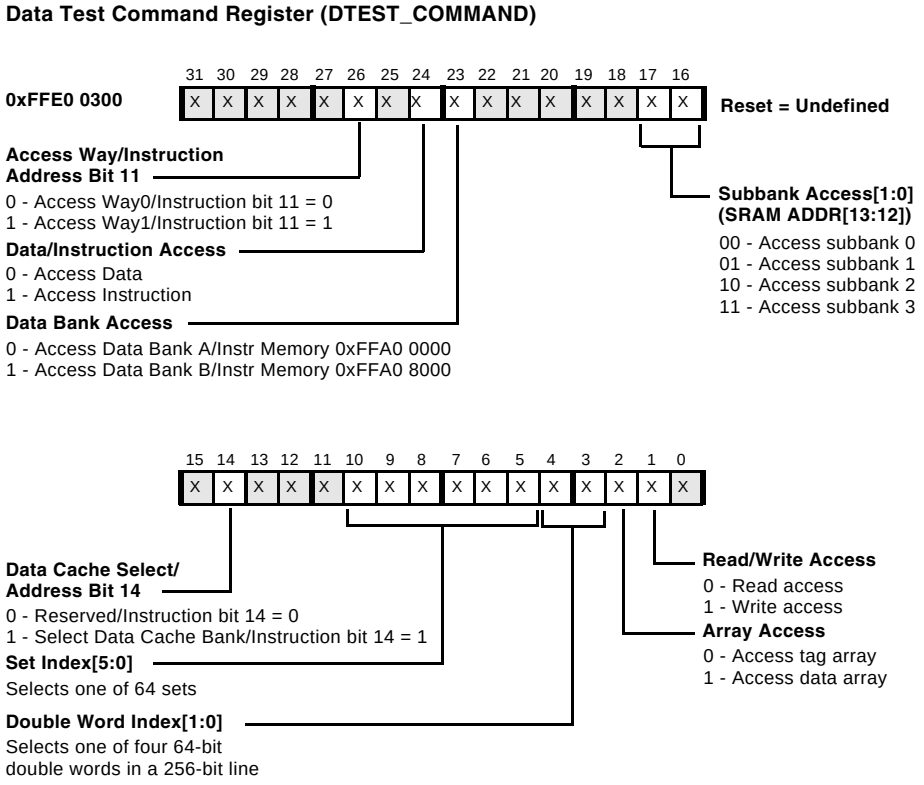

Data Test Command Register (DTEST_COMMAND)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0xFFE0 0300
Reset = Undefined
Access Way/Instruction Address Bit 11
0 - Access Way0/Instruction bit 11 = 0
1 - Access Way1/Instruction bit 11 = 1
Data/Instruction Access
0 - Access Data
1 - Access Instruction
Data Bank Access
0 - Access Data Bank A/Instr Memory 0xFFA0 0000
1 - Access Data Bank B/Instr Memory 0xFFA0 8000
Subbank Access[1:0] (SRAM ADDR[13:12])
00 - Access subbank 0
01 - Access subbank 1
10 - Access subbank 2
11 - Access subbank 3
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Data Cache Select/ Address Bit 14
0 - Reserved/Instruction bit 14 = 0
1 - Select Data Cache Bank/Instruction bit 14 = 1
Set Index[5:0]
Selects one of 64 sets
Double Word Index[1:0]
Selects one of four 64-bit double words in a 256-bit line
Read/Write Access
0 - Read access
1 - Write access
Array Access
0 - Access tag array
1 - Access data array

Figure 3-2. Data Test Command Register

4 ONE-TIME PROGRAMMABLE MEMORY

This chapter describes One-Time-Programmable (OTP) memory features of the ADSP-BF52x Blackfin processor.

The chapter includes the following sections:

- "OTP Memory Map" on page 4-2
- "Error Correction" on page 4-7
- "OTP Access" on page 4-10
- "Error Correction Policy" on page 4-8
- "OTP Timing Parameters" on page 4-12
- "Callable ROM Functions for OTP ACCESS" on page 4-17
- "Programming and Reading OTP" on page 4-19
- "Write-protecting OTP Memory" on page 4-25
- "Accessing Private OTP Memory" on page 4-28
- "OTP Programming Examples" on page 4-28

OTP Memory Overview

The ADSP-BF52x processors include an on-chip, one-time-programmable memory array which provides 64k-bits of non-volatile memory. This includes the array and logic to support read access and programming. A

mechanism for error correction is also provided. Additionally, pages can be write protected.

OTP memory can be programmed through various methods, including software running on the Blackfin processor. The ADSP-BF52x processors provide C and assembly callable functions in the on-chip ROM to help the developer access the OTP memory.

The one-time-programmable memory (OTP) is divided into two main regions. A 32K bit “public” unsecured region, which has no access restrictions, and a 32K bit “private” secured region with access restricted to authenticated code when operating in Secure Mode (For information about these modes, see Chapter 16, “Security” of the *ADSP-BF52x Blackfin Processor Hardware Reference.*)

OTP enables developers to store both public and private data on-chip. A 64K x 1 bit array is available as shown in Figure 4-2 on page 4-4 and Figure 4-3 on page 4-5. In addition to storing public and private data, it allows developers to store completely user-definable data, such as customer ID, product ID, MAC address, etc.

The public portion of OTP memory contains many “factory set only” values. Users are urged to exercise caution when writing to OTP memory and to consult the OTP memory map for details of Customer Programmable Settings (CPS) and factory reserved areas of this memory. See also Factory Page Settings (FPS) and Preboot Page Settings (PBS) in “System Reset and Booting” on page 17-1 in this volume of the *ADSP-BF52x Blackfin Processor Hardware Reference*.

OTP Memory Map

The OTP is not part of the Blackfin linear memory map. It has a separate memory map that is shown in Figure 4-2 on page 4-4 and Figure 4-3 on page 4-5. OTP memory is not accessed directly using the Blackfin mem-

ory map; rather, it is accessed via four 32-bit wide registers (OTP_DATA3–0) which act as the OTP memory read/write buffer.

In the case of an OTP memory read, the OTP_DATAx registers will contain the 16-byte result of the OTP memory access. In the case of an OTP memory write, the OTP_DATAx registers will contain 16 bytes of data to be written to the OTP memory.

OTP_DATA3–0 registers are organized into a 128 bit page as shown in Figure 4-1.

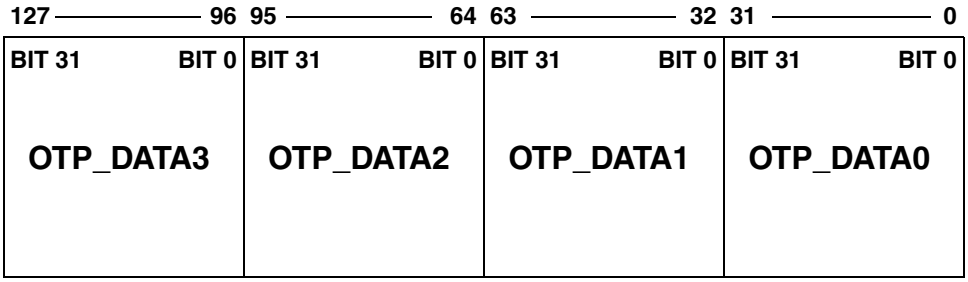


Figure 4-1. OTP_DATAx Registers

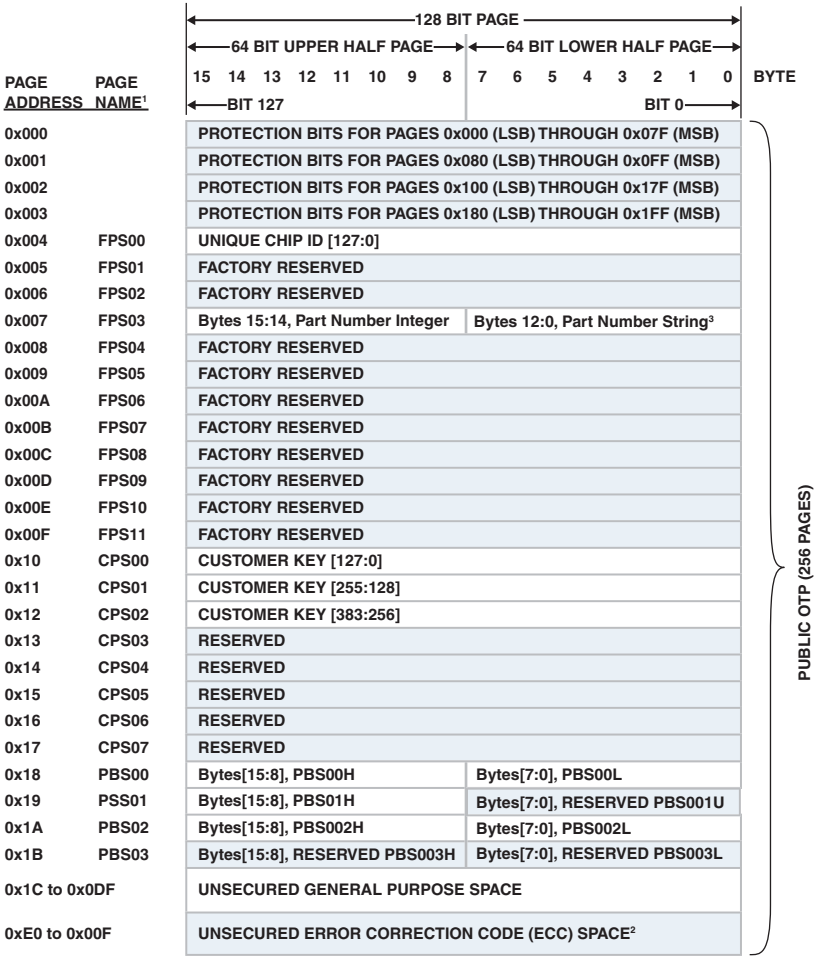
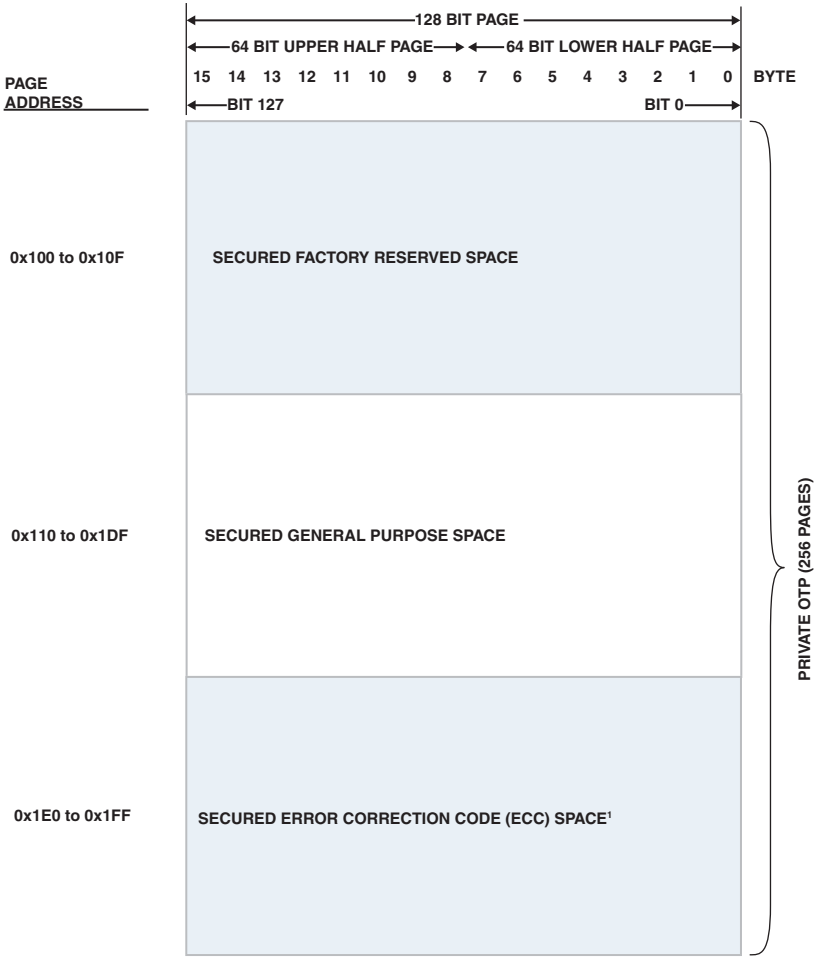

128 BIT PAGE
64 BIT UPPER HALF PAGE
64 BIT LOWER HALF PAGE
15 14 13 12 11 10 9 8
7 6 5 4 3 2 1 0
BYTE
BIT 127
BIT 0
PAGE ADDRESS
PAGE NAME1
0x000 PROTECTION BITS FOR PAGES 0x000 (LSB) THROUGH 0x07F (MSB)
0x001 PROTECTION BITS FOR PAGES 0x080 (LSB) THROUGH 0x0FF (MSB)
0x002 PROTECTION BITS FOR PAGES 0x100 (LSB) THROUGH 0x17F (MSB)
0x003 PROTECTION BITS FOR PAGES 0x180 (LSB) THROUGH 0x1FF (MSB)
0x004 FPS00 UNIQUE CHIP ID [127:0]
0x005 FPS01 FACTORY RESERVED
0x006 FPS02 FACTORY RESERVED
0x007 FPS03 Bytes 15:14, Part Number Integer | Bytes 12:0, Part Number String3
0x008 FPS04 FACTORY RESERVED
0x009 FPS05 FACTORY RESERVED
0x00A FPS06 FACTORY RESERVED
0x00B FPS07 FACTORY RESERVED
0x00C FPS08 FACTORY RESERVED
0x00D FPS09 FACTORY RESERVED
0x00E FPS10 FACTORY RESERVED
0x00F FPS11 FACTORY RESERVED
0x10 CPS00 CUSTOMER KEY [127:0]
0x11 CPS01 CUSTOMER KEY [255:128]
0x12 CPS02 CUSTOMER KEY [383:256]
0x13 CPS03 RESERVED
0x14 CPS04 RESERVED
0x15 CPS05 RESERVED
0x16 CPS06 RESERVED
0x17 CPS07 RESERVED
0x18 PBS00 Bytes[15:8], PBS00H | Bytes[7:0], PBS00L
0x19 PSS01 Bytes[15:8], PBS01H | Bytes[7:0], RESERVED PBS001U
0x1A PBS02 Bytes[15:8], PBS002H | Bytes[7:0], PBS002L
0x1B PBS03 Bytes[15:8], RESERVED PBS003H | Bytes[7:0], RESERVED PBS003L
0x1C to 0x0DF UNSECURED GENERAL PURPOSE SPACE
0xE0 to 0x00F UNSECURED ERROR CORRECTION CODE (ECC) SPACE2
PUBLIC OTP (256 PAGES)

Figure 4-2. Public OTP Memory Map[1, 2, 3]

1 Factory Programmable Settings (FPS) are programmed at the factory. Customer Programmable Settings (CPS) are programmed by the customer.

2 This space should NOT be written by the customer. 8-bit error correction codes are automatically generated by firmware and stored in this region.

3 Part Number Field Definition.


128 BIT PAGE
64 BIT UPPER HALF PAGE
64 BIT LOWER HALF PAGE
PAGE ADDRESS
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
BYTE
BIT 127
BIT 0
0x100 to 0x10F
SECURED FACTORY RESERVED SPACE
0x110 to 0x1DF
SECURED GENERAL PURPOSE SPACE
0x1E0 to 0x1FF
SECURED ERROR CORRECTION CODE (ECC) SPACE1
PRIVATE OTP (256 PAGES)

Footnotes
1. This space should NOT be written by the customer. 8-bit error correction codes are automatically generated by firmware a stored in this region.

Figure 4-3. Private OTP Memory Map

Part Number Field Definition. A string indicating the model number of the product is programmed into FPS03 (see Figure 4-2 on page 4-4. Each character is represented by standard 8-bit ASCII code. A termination character of 0x0000 0000 terminates the string. The field supports up to

12 alphanumeric characters plus one termination character. The first string character resides in bits[7:0] and the string grows to the left with the left most character being the termination character. Integer representation of the part number is shown in Table 4-1. Byte 13 in FPS03 is reserved.

Table 4-1. Part Number Field Definition

Part Number	CODEC	Code
ADSP-BF522	N/A	0x020A
ADSP-BF523	N/A	0x020B
ADSP-BF524	N/A	0x020C
ADSP-BF525	N/A	0x020D
ADSP-BF526	N/A	0x020E
ADSP-BF527	N/A	0x020F
ADSP-BF522	C1	0x820A
ADSP-BF523	C1	0x820B
ADSP-BF524	C1	0x820C
ADSP-BF525	C1	0x820D
ADSP-BF526	C1	0x820E
ADSP-BF527	C1	0x820F
ADSP-BF522	C2	0x420A
ADSP-BF523	C2	0x420B
ADSP-BF524	C2	0x420C
ADSP-BF525	C2	0x420D
ADSP-BF526	C2	0x420E
ADSP-BF527	C2	0x420F

OTP memory ranges marked as Factory Reserved, Reserved and Error Correction Code Space (see Figure 4-2 on page 4-4) must not be pro-

grammed by the user. Customer Programmable Settings are optionally programmed by the developer.

Page-Protection bits provide protection for each 128-bit page within the OTP. By default, the OTP array bits are not set and will read back as zero values if left unprogrammed. Programmed data values consist of zeroes and ones; therefore, after programming OTP memory, some bits will intentionally remain as zero values. The write-protect bits provide protection for the zero value bits to remain as zeroes and prevent future programming (inadvertent or malicious) from changing bit values from zero to one.

Pages 0x10, 0x11, and 0x12 hold the customer public key, which is used for Lockbox digital signature authentication. Please refer to Chapter 16, "Security" for more information on Lockbox and how the public key is used.

OTP memory is logically arranged in a sequential set of 128-bit pages. Each OTP memory address refers to a 128-bit page. The ADSP-BF52x processor thus provides 512 pages of OTP memory.

In order to read or program the OTP memory, a set of functions are provided in the on-chip ROM. These functions include `bfrom_OtpRead()`, `bfrom_OtpWrite()` and `bfrom_OtpCommand()`.

Error Correction

To meet strict quality goals, error correction is used to ensure data integrity. `bfrom_OtpRead()` and `bfrom_OtpWrite()`, provided in the on-chip ROM, support error correction.

Error correction works by calculating an 8-bit Error Correction Code (ECC) for each 64-bit data word (half page) when it is programmed into the OTP. When this word is later read from OTP, its corresponding ECC is also read, and a data integrity check is performed. If the check fails,

error correction on the data word can be attempted using the ECC. Depending on the type of error, the error correction algorithm will perform as shown in Table 4-2.

Table 4-2. Hamming Code Single Error Corrections, Double Error Detection

Number of bad bits in data word	Error(s) detected?	Error(s) corrected?
0	N/A	N/A
1	Yes	Yes
2	Yes	No
3 or more	No	No

Error Correction Policy

1. Error correction requires that OTP space is written and read in 64-bit widths. Firmware will only support writing or reading half of an OTP page.
2. Error correction is used to correct data in all pages of OTP space, except the protection pages (0x0 to 0x3) and ECC pages themselves. See "OTP Access" on page 4-10 for more information.
3. Firmware will generate and program the 8-bit ECC fields as mapped in Table 4-3 and Table 4-4.
4. The developer is responsible for locking both the data page(s) *and* the ECC page(s) after all programming is complete.
5. Pages 0x04 to 0x0F are reserved for ADI factory use. Therefore, pages 0x004 to 0x00F, 0x0E0, and 0x0E1 will be locked coming out of the Analog Devices factory.

Table 4-3. Mapping for Storage of Error Correction Codes for Unsecured OTP Space

Page	Byte							
	15	14	13	12	11	10	9	8
0x0E0	0x007U	0x007L	0x006U	0x006L	0x005U	0x005L	0x004U	0x004L
0x0E1	0x00FU	0x00FL	0x00EU	0x00EL	0x00DU	0x00DL	0x00CU	0x00CL
0x0E2	0x017U	0x017L	0x016U	0x016L	0x015U	0x015L	0x014U	0x014L
....								
0x0FB	0x0DFU	0x0DFL	0x0DEU	0x0DEL	0x0DDU	0x0DDL	0x0DCU	0x0DCL
Page	7	6	5	4	3	2	1	0
0x0E0	Unused	Unused	Unused	Unused	Unused	Unused	Unused	Unused
0x0E1	0x00BU	0x00BL	0x00AU	0x00AL	0x009U	0x009L	0x008U	0x008L
0x0E2	0x013U	0x013L	0x012U	0x012L	0x011U	0x011L	0x010U	0x010L
....								
0x0FB	0x0DBU	0x0DBL	0x0DAU	0x0DAL	0x0D9U	0x0D9L	0x0D8U	0x0D8L

Table 4-4. Mapping for Storage of Error Correction Codes for Secured OTP Space

Page	Byte							
	15	14	13	12	11	10	9	8
0x1E0	0x107U	0x107L	0x106U	0x106L	0x105U	0x105L	0x104U	0x104L
0x1E1	0x10FU	0x10FL	0x10EU	0x10EL	0x10DU	0x10DL	0x10CU	0x10CL
0x1E2	0x117U	0x117L	0x116U	0x116L	0x115U	0x115L	0x114U	0x114L
....								
0x1FB	0x1DFU	0x1DFL	0x1DEU	0x1DEL	0x1DDU	0x1DDL	0x1DCU	0x1DCL
Page	7	6	5	4	3	2	1	0
0x1E0	0x103U	0x103L	0x102U	0x102L	0x101U	0x101L	0x100U	0x100L
0x1E1	0x10BU	0x10BL	0x10AU	0x10AL	0x109U	0x109L	0x108U	0x108L
0x1E2	0x113U	0x113L	0x112U	0x112L	0x111U	0x111L	0x110U	0x110L
....								
0x1FB	0x1DBU	0x1DBL	0x1DAU	0x1DAL	0x1D9U	0x1D9L	0x1D8U	0x1D8L

OTP Access

The ADSP-BF52x on-chip ROM contains functions for initializing OTP timing parameters, and for reading and programming the OTP memory. These functions include `bfrom_OtpRead()`, `bfrom_OtpWrite()` and `bfrom_OtpCommand()`.

These functions are callable from C or assembly application code. Use only these functions for accessing OTP memory. Directly accessing memory locations within OTP memory by other means is not supported.

The existing ECC in ROM is known as “Hamming [72,64]”. This is specifically a 64-bit data, +8-bit ECC field, for 1-bit correction and 2-bit error detection scheme.

The ROM-based OTP read/write API *must* be used for all OTP data accesses (see limited exceptions below). ADI does not support any ECC other than the ECC provided by ADI within the ROM API. This is the *only* ECC method supported by Analog Devices. Analog Devices does not support direct access of OTP data without using error correction. All attempts to implement other schemes are not guaranteed or supported by Analog Devices.

Exceptions: The only bits that do not use ECC are page lock bits (first 4 pages) and the preboot invalidate bits. See the Preboot section in “System Reset and Booting” on page 17-1.

OTP memory programming is done serially under software control. Since the unprogrammed OTP memory value defaults to zero—only bits whose value is intended to be “1” have to be programmed. In order to protect areas of OTP memory that have been programmed, or areas which have intentionally been left unprogrammed which end users wish to remain unchanged, write-protect bits can be set for each 128-bit page within OTP memory. Each write-protect bit, when set, will prevent further programming attempts to OTP memory on a per page basis. Refer to the OTP memory map (Figure 4-2 on page 4-4) for details.

The ADSP-BF52x Blackfin processor can program OTP through software code executing directly on the Blackfin processor. For the ADSP-BF523/525/527 processor, a charge pump residing on-chip is used to apply the voltage levels appropriate for programming OTP memory. For the ADSP-BF522/524/526 processor, no on-chip charge pump exists, and an externally applied voltage is required to apply the voltage levels appropriate for programming OTP memory. Refer to the processor data sheet for `VPPOTP` specifications. OTP programming code can be loaded into the processor via JTAG emulation, DMA, and all supported boot methods.

OTP memory can only be written once (changing a bit from 0 to 1). Once a bit has been changed from a 0 to a 1, it cannot be changed back to 0. The write-protect bits prevent OTP memory that has already been programmed from having any bits that are meant to remain as 0 value later programmed to a value of 1.

Prior to accessing OTP memory, refer to the product data sheet for specifications on `VDDOTP` and `VPPOTP` voltage levels to ensure reliable OTP programming. OTP timing parameter settings must be set prior to attempting any write accesses to OTP.

OTP Timing Parameters

In order to read and program the OTP memory reliably, set the OTP timing parameters prior to accessing OTP memory. All of the timing parameters are fields within the `OTP_TIMING` register (see "OTP_TIMING Register" on page 4-17). The `bfrom_OtpCommand()` function (detailed in the following sections) is provided in the on-chip ROM to program the timing parameters.

The OTP timing parameters must be set using the `bfrom_OtpCommand()`.

OTP read accesses can use the OTP timing default reset value (`OTP_TIMING` = 0x0000 1485).

Using the OTP timing default reset value for writes will result in write errors as this timing value is not appropriate for performing write accesses.

Insufficient voltage/current provided to OTP during write access or incorrect OTP timing parameters may return error code 0x11 (multiple bad bits in 64-bit data) during OTP writes. Subsequent reads from this page return 0.

Timing for the ADSP-BF523/525/527 Processors

The OTP timing parameters consist of several concatenated fields that form one value, which is passed as an argument to the `bfrom_OtpCommand()` function. The developer must calculate a value based upon the desired `SCLK` frequency at which the OTP access will be performed. This calculated value is combined with a constant value provided by ADI to arrive at the appropriate access setting.

The OTP timing parameters are comprised of two values:

`OTP_TIMING[7:0]` = `OTP_TP1` = 1000/sclk_period (in nanoseconds)

`OTP_TIMING[31:8]` = `OTP_TP2` = 0x0A5488

To ensure reliable OTP write accesses, the user-calculated field must be combined with the `OTP_TP2` field specified by Analog Devices as shown in Listing 4-1 and Listing 4-2.

Example calculations shown in Listing 4-1 and Listing 4-2 are based upon the `VDDOTP` and `VPPOTP` voltages specified in the *ADSP-BF522/523/524/525/526/527 Blackfin Embedded Processor Data Sheet*. The OTP timing parameter calculations are dependent upon the user-defined `SCLK` frequency. Do not rely on the specifications in the examples—refer to the processor data sheet for actual `VDDOTP` voltage, `VPPOTP` voltage, and `SCLK` specifications.

For `SCLK` = 12.5ns (80 MHz), the following calculations determine the OTP timing argument for the `bfrom_OtpCommand()` call.

OTP_TP1 = 1000/sclk_period = 1000/12.5 = 0x50	0x0000 0050
OTP_TP2 = (constant)	0x0A54 88xx
Calculated OTP timing parameter value	0x0A54 8850

The code for the API call (in C) is:

Listing 4-1. OTP Timing Calculations for SCLK = 80 MHz

```
// Initialize OTP access settings
// Proper access settings for SCLK = 80 MHz
const u32 OTP_init_value = 0x0A548850;
return_code = bfrom_OtpCommand( OTP_INIT, OTP_init_value);
```

For SCLK = 20.0ns (50 MHz), the following calculations determine the OTP timing argument for the bfrom_OtpCommand() call.

OTP_TP1 = 1000/sclk_period = 1000/20.0 = 0x32	0x0000 0032
OTP_TP2 = (constant)	0x0A54 88xx
Calculated OTP timing parameter value	0x0A54 8832

The code for the API call (in C) is:

Listing 4-2. OTP Timing Calculations for SCLK = 50 MHz

```
// Initialize OTP access settings
// Proper access settings for SCLK = 50 MHz
const u32 OTP_init_value = 0x0A548832;
return_code = bfrom_OtpCommand( OTP_INIT, OTP_init_value);
```

Timing for the ADSP-BF522/524/526 Processors

The OTP timing parameters consist of several concatenated fields that form one value, which is passed as an argument to the bfrom_OtpCommand() function. The developer must calculate a value based upon the desired SCLK frequency at which the OTP access will be performed. This calculated value is combined with a constant value provided by ADI to arrive at the appropriate access setting.

The OTP timing parameters are comprised of two values:

OTP_TIMING[7:0] = OTP_TP1 = 1000/sclk_period (in nanoseconds)

OTP_TIMING[31:8] = OTP_TP2 = 0x145487

To ensure reliable OTP write accesses, the user-calculated field must be combined as shown in Listing 4-3 and Listing 4-4 with the OTP_TP2 field specified by Analog Devices.

Example calculations in Listing 4-3 and Listing 4-4 are based upon the VDDOTP and VPPOTP voltages specified in the *ADSP-BF522/523/524/525/526/527 Blackfin Embedded Processor Data Sheet*. The OTP timing parameter calculations are dependent upon the user-defined SCLK frequency. Do not rely on the specifications in the examples—refer to the processor data sheet for actual VDDOTP voltage, VPPOTP voltage, and SCLK specifications.

For SCLK = 12.5ns (80 MHz), the following field calculations determine the OTP timing argument for the bfrom_OtpCommand() call.

OTP_TP1 = 1000 / sclk_period = 1000 / 12.5 = 0x50	0x00000050
OTP_TP2 = (constant)	0x145487xx
Calculated OTP timing parameter value	0x14548750

The code for the API call (in C) is:

Listing 4-3. OTP Timing Calculations for SCLK = 80 MHz

```
// Initialize OTP access settings
// Proper access settings for SCLK = 80 MHz
const u32 OTP_init_value = 0x14548750;
return_code = bfrom_OtpCommand( OTP_INIT, OTP_init_value);
```

For SCLK = 20.0ns (50 MHz), the following field calculations determine the OTP timing argument for the bfrom_OtpCommand() call.

OTP_TP1 = 1000 / sclk_period = 1000 / 20.0 = 0x32	0x00000032
OTP_TP2 = (constant)	0x145487xx
Calculated OTP timing parameter value	0x14548732

The code for the API call (in C) is:

Listing 4-4. OTP Timing Calculations for SCLK = 50 MHz

```
// Initialize OTP access settings
// Proper access settings for SCLK = 50 MHz
const u32 OTP_init_value = 0x14548732;
return_code = bfrom_OtpCommand( OTP_INIT, OTP_init_value);
```

OTP_TIMING Register

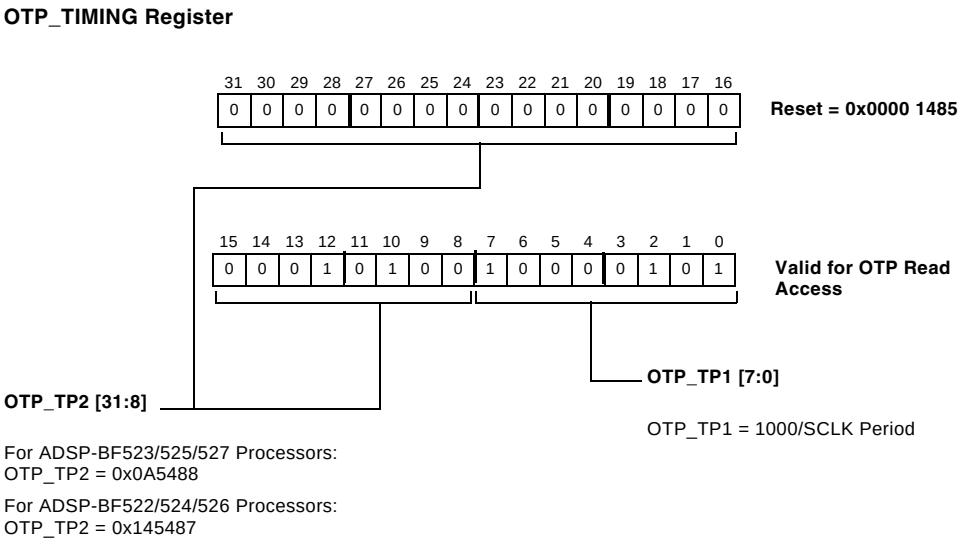


Figure 4-4. OTP_TIMING Register

Callable ROM Functions for OTP ACCESS

The following functions support OTP access.

Initializing OTP

This section describes the `bfrom_OtpCommand()` function for OTP memory controller setup. The prototype and macros to decode the function's return codes are supplied in the `bfrom.h` header file located in the VisualDSP++ installation directory. The meaning of the error code is described in "Error Codes" on page 4-24.

bfrom_OtpCommand

This function implements various commands to setup the OTP controller. The first parameter is a mnemonic label specifying the command. The

second parameter is a generic value passed as the argument for the requested command. The second parameter is optional and can be an integer value or (via opportune casting) a pointer to an extension structure. There are two commands:

- `OTP_INIT`
  sets the required timing value (register `OTP_TIMING`) to “value”
- `OTP_CLOSE`
  reinitializes the OTP controller.
  This can be called by the user before exiting Secure Mode if desired. The value parameter may be specified as 0 or NULL.

Entry address: 0xEF00 0018

Arguments:

`R0`: command (`dCommand`)
`OTP_INIT`
`OTP_CLOSE`

`R1`: timing value to be programmed (dValue), not used for `OTP_CLOSE`

C Prototype: `u32 bfrom_OtpCommand(u32 dCommand, u32 dValue);`

Return code:

`bfrom_OtpCommand()` currently always returns with “0”.

In the previous examples, the OTP timing parameter was calculated to be 0x0A54 8850 for the ADSP-BF527 processor with `SCLK` = 80 MHz. See Listing 4-1 on page 4-14. Listing 4-5 on page 4-18 shows a sample of C code that uses the `bfrom_OtpCommand()` function to program this timing parameter.

Listing 4-5. Example Use of `bfrom_OtpCommand()`

```
#include <bfrom.h>
```

```
#define OTP_TIMING_PARAM (0x0A548850)
u32 Otp_Timing_Param_Init()
{
   u32 otp_timing_parameter;
   u32 RetVal;
   otp_timing_parameter = OTP_TIMING_PARAM;
   RetVal = bfrom_OtpCommand(OTP_INIT, otp_timing_parameter);
   // (equivalently, with a variable):
   RetVal = bfrom_OtpCommand(OTP_INIT, OTP_TIMING_PARAM);
      return RetVal;
}
```

More examples:

Listing 4-6. Another Example Use of `bfrom_OtpCommand()`

```
//timing parameter
const u32 init_value = 0x0A548850;
// call sets OTP_TIMING register
RetVal = bfrom_OtpCommand(OTP_INIT, init_value);
// call sets OTP_TIMING register
RetVal = bfrom_OtpCommand(OTP_INIT, 0x0A548850);
// call clears OTP controller and data registers
RetVal = bfrom_OtpCommand(OTP_CLOSE, NULL);
```

The prototype of `bfrom_OtpCommand()` is also included in the `bfrom.h` header file installed with VisualDSP++ 5.0 and later releases. The `OTP_INIT` macro is defined in `bfrom.h` as well.

Programming and Reading OTP

This section describes the `bfrom_OtpRead()` and `bfrom_OtpWrite()` functions provided in the ADSP-BF52x processor's on-chip ROM. The prototypes and macros to decode the return codes are supplied in the

bfrom.h header file located in the VisualDSP++ installation directory. The meaning of the error code is described in "Error Codes" on page 4-24.

bfrom_OtpRead

This function reads 64-bit OTP half-pages using error correction.

Entry address: 0xEF00 001A

Arguments:

R0: OTP page address (dPage)

R1: Flags (dFlags)
```
OTP_LOWER_HALF
OTP_UPPER_HALF
OTP_NO_ECC
```

R2: Pointer to 64-bit memory struct (long long) to put read data (*pPageContent)

C prototype: `u32 bfrom_OtpRead (u32 dPage, u32 dFlags, u64 *pPageContent);`

Return code:

R0: error or warning code, see Table 4-5.

This function reads a half-page and stores the content in the 64-bit variable pointed to by its last parameter. The page parameter defines the address. The flags parameter defines whether the upper or the lower half page is to be read. The default reset OTP_TIMING value may be used for all read accesses without requiring that a new value be programmed prior to read accesses. Programming a value valid for write accesses will also allow read accesses.

Flag parameter OTP_NO_ECC should not be used with any OTP read access as it bypasses error correction code support. It is only for diagnostic use.

bfrom_OtpWrite

This function writes a half-page with the content in the 64-bit variable pointed to by its last parameter. The page parameter defines the address.

Entry address: 0xEF00 001C

Arguments:

`R0`: OTP page address (`dFlag`)

`R1`: Flags (`dFlags`)

```
OTP_LOWER_HALF
OTP_UPPER_HALF
OTP_NO_ECC
OTP_LOCK
OTP_CHECK_FOR_PREV_WRITE
```

R2: Pointer to 64-bit memory struct (long long) that contains the data to be written to OTP memory (`*pPageContent`)

C Prototype: `u32 bfrom_OtpWrite (u32 dPage, u32 dFlags, u64 *pPageContent);`

Return code:

`R0`: error or warning code, see Table 4-5.

The `dFlags` parameter defines whether the upper or the lower half page is to be written and whether the target half page should be checked for a previously written value before any write attempt is made. Additionally, a page can optionally be locked (permanently protected against further writes).

When performing pure lock operations (only locking a page without writing any data values to it), the half-page parameter is not required and it makes no difference which half-page is specified if this parameter is included in the function call.

To reduce the probability of inadvertent writes to OTP pages, this function checks for a valid OTP write timing setting in the OTP_TIMING register. Specifically, bits [31:15] must not be equal to zero. When this field is equal to zero, calls to the write routine cause an access violation error and the requested action is not performed. By calling the bfrom_OtpCommand (OTP_INIT, …) function with appropriate values for reads only and for read/write accesses—the user can protect against inadvertent writes. Users are free to ignore this mechanism by calling bfrom_OtpCommand (OTP_INIT, …) only once for read/write access.

When the flag OTP_CHECK_FOR_PREV_WRITE is *not* specified, a previously written value will be overwritten, both in the ECC and in the data fields, for any unlocked page where a write access is performed. Of course, once a bit was set to "1" it cannot be reset to "0" by the new write operation. This means that, in all likelihood, if the new value is different from the previous one, the result will have multiple bit errors, in either or both the ECC and data fields.

Since the ECC field is written first by the ROM function, a multiple bit error will abort the operation without writing the new data value to the OTP data page.

Since multiple bit errors have a statistical chance of not being detected as such, this default mode of operation should not be used. Or used with appropriate caution.

The flag OTP_CHECK_FOR_PREV_WRITE should always be used when performing write accesses to OTP with the bfrom_OtpWrite() function.

If the flag OTP_CHECK_FOR_PREV_WRITE is specified in the call, a write to a previously programmed page causes dedicated error messages and will not be performed. Errors are generated as follows. The 64-bit data and the 8-bit ECC field are read and the total number of "1"s is counted. If this number is equal to or greater than two, the error flag OTP_PREV_WR_ERROR is returned and the write operation is not performed. If the number is 0, the page is certainly blank and the write is performed. If the number is

one, a more thorough check is performed. If the “1” is in the ECC field, an error flag OTP_SB_DEFECT_ERROR is returned and the write is not performed. If the “1” is in the data field, it is determined whether the value to be written contains a “1” in the same position. If so, the write is performed. If not, the error flag OTP_SB_DEFECT_ERROR is returned and the write is not performed. This error code warns the user that a single-bit defect could be in the page. The user can then decide whether to use this page regardless (by repeating the call without the OTP_CHECK_FOR_PREV_WRITE flag) or skip this page.

The OTP_CHECK_FOR_PREV_WRITE flag is ignored when a pure lock operation is requested (for example, a OTP_LOCK flag is set and *pPageContent = NULL). It is therefore unnecessary and harmless to specify this flag.

The OTP_CHECK_FOR_PREV_WRITE flag is not ignored when doing a lock operation after a write (for example, OTP_LOCK + write in the same call and *pPageContent = NULL).

If the flag parameter for the write operation is OR’d with the OTP_LOCK flag; the write operation, if successful, will be immediately followed by setting the protection bit for the requested full 128-bit page.

A special case for using (OTP_LOCK) is the following. If the third parameter is NULL, this call will lock a page without writing any data value to it (pure lock function). Note that in this case, “page” can span all pages from 0x000 to 0x1FF. *This is the only way to lock the ECC pages themselves.*

The use of flag parameter OTP_NO_ECC is only supported in write operations when used to implement write-protection/page-locking. Bypassing error correction in OTP writes may result in loss of OTP data integrity and is not supported for any other OTP access. The preferred method of locking pages is to use the OTP_LOCK parameter in the bfrom_Otp_Write

function (see “Write-protecting OTP Memory” on page 4-25. Or the pre-boot invalidate bits can be set (see “Preboot” on page 17-11).

The use of ECC in all OTP accesses other than in this limited exception is mandatory.

Error Codes

This section describes the returned error codes from the API functions.

Returned Error Codes from API Functions

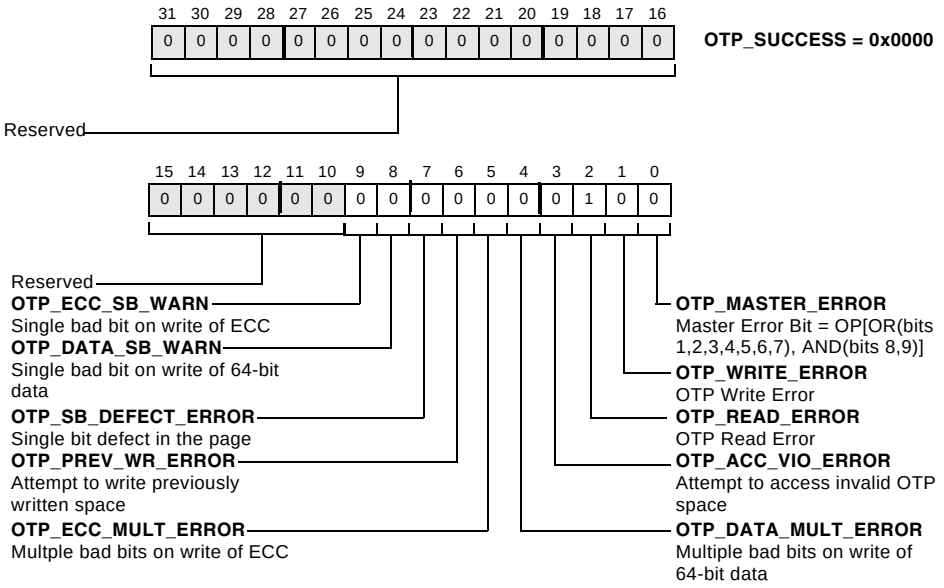

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
OTP_SUCCESS = 0x0000
Reserved
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
Reserved
OTP_ECC_SB_WARN
Single bad bit on write of ECC
OTP_DATA_SB_WARN
Single bad bit on write of 64-bit data
OTP_SB_DEFECT_ERROR
Single bit defect in the page
OTP_PREV_WR_ERROR
Attempt to write previously written space
OTP_ECC_MULT_ERROR
Multple bad bits on write of ECC
OTP_MASTER_ERROR
Master Error Bit = OP[OR(bits 1,2,3,4,5,6,7), AND(bits 8,9)]
OTP_WRITE_ERROR
OTP Write Error
OTP_READ_ERROR
OTP Read Error
OTP_ACC_VIO_ERROR
Attempt to access invalid OTP space
OTP_DATA_MULT_ERROR
Multiple bad bits on write of 64-bit data

Figure 4-5. Returned Error Codes from API Functions

Table 4-5. Returned Error Codes from API Functions

Bit Position	Name	Example Return Value	Definition
N/A	OTP_SUCCESS	0x0	No Error
0	OTP_MASTER_ERROR	0x1	Master Error Bit = OR [OR (bits 1,2,3,4,5,6,7), AND (bits 8,9)]
1	OTP_WRITE_ERROR	0x3	(E) OTP Write Error
2	OTP_READ_ERROR	0x5	(E) OTP Read Error
3	OTP_ACC_VIO_ERROR	0x9	(E) Attempt to access invalid OTP space
4	OTP_DATA_MULT_ERROR	0x11	(E) Multiple bad bits on write of 64 bit data
5	OTP_ECC_MULT_ERROR	0x21	(E) Multiple bad bits on write of ECC
6	OTP_PREV_WR_ERROR	0x41	(E) Attempt to write previously written space
7	OTP_SB_DEFECT_ERROR	0x81	(E) Single-bit defect in the page
8	OTP_DATA_SB_WARN	0x100	(W) Single bad bit on write of 64 bit data
9	OTP_ECC_SB_WARN	0x200	(W) Single bad bit on write of ECC

`bfrom_OtpRead()` returns with an error when any of the bits [6:2] are set or both bits [9:8] are set. In this case, the `OTP_MASTER_ERROR` bit is also set. It returns with a warning if only one of the bits [9:8] is set.

`bfrom_OtpWrite()` returns with an error when any of the bits [7:1] are set or both bits [9:8] are set. In this case, the `OTP_MASTER_ERROR` bit is also set. It returns with a warning if only one of the bits [9:8] is set.

`bfrom_OtpCommand()` currently always returns with “0”.

Write-protecting OTP Memory

As shown in Figure 4-2 on page 4-4, a small portion of OTP memory is reserved for write-protect bits (“write-protect” is synonymous with

"page-protect" in the context of this discussion). After programming OTP memory, the programmer can use these protection bits to "lock" the page that was just programmed by setting the write-protect bit corresponding to the OTP data page. Once the write-protect bit is set and the lock is in place, further attempts to write to that page are not allowed, resulting in an error. Page protect bits also can be set to prevent programming of unwritten OTP pages. Once an OTP page is page-protected, the write protection can not be reversed and no further write accesses can be made to the protected page(s).

There are four pages reserved for the write-protection bits. Page 0x0 through page 0x3 contain the 512 write-protect bits, one bit for each of the 512 data pages within OTP memory. The first two write-protect bit pages (page 0x0 and page 0x1) correspond to the public (non-secure) regions of the OTP map. The other two write-protect bit pages (page 0x2 and page 0x3) correspond to the protection of private (secure) regions of the OTP map. The processor does not need to be operating in Secure Mode in order to program protection pages associated with secure OTP regions. All protection bits can be written in any security state including Open Mode.

While reads and writes access a half-page at a time, setting a protection bit for a page will effectively lock an entire page for future write accesses (lower and upper half page). The programmer must ensure that all required programming is completed on a full 128-bit OTP data page before setting the write-protect bit for that page. Before locking the page, the programmer must make sure that the full 128-bit OTP page is programmed, or that no future programming will be required for the unprogrammed portion of the page.

If P is the OTP page that is to be write-protected, the write-protect bit and its page can be calculated as follows.

Let `WPP` be the write-protect page where the write-protect bit resides.
Let `WPB` be the write-protect bit that needs to be set to lock page P.

The write-protect page can be calculated by:

```
WPP = P >> 7;
```

and the write-protect bit can be calculated by:

```
WPB = P & 0x7f;
```

Manual calculation generally unnecessary since the `bfrom_OtpWrite()` function can be used to lock pages (see "OTP Programming Examples" on page 4-28).

```
// lock page (note third parameter equals NULL)
return_code = bfrom_OtpWrite( 0x01C, OTP_LOCK, NULL);
```

Locking a single ECC (error correction code) page results in locking the correction codes, which correspond to eight OTP data pages (16 half pages). This is because a 64-bit half-page access must be performed when write protecting the ECC page and every 8-bits within an ECC page is a parity correction code which corresponds to a 64-bit half-page of data in OTP. Therefore, a full 128-bit ECC page holds the correction codes for eight full 128-bit pages of data in OTP, or 16 half-pages. Pages can only be locked as full 128-bit pages even though read/write accesses may occur at 64-bit half-page granularity. Locking a single ECC page will prevent further write access to the corresponding eight OTP data pages.

ECC (error correction code) space is not permitted to be written to directly.

For example, locking ECC page 0xFB will result in locking the error correction parity data associated with the 16 data pages in the range 0x0D8–0x0DF.

```
// Only Lock ECC code page
```

```
return_code = bfrom_OtpWrite(0xFB, OTP_LOCK, NULL);
```

No further write accesses to the ECC page 0xFB or corresponding data pages 0x0D8–0x0DF will be allowed after write protection of the ECC page in this example.

Bits [3:0] of OTP page 0 are the write-protect bits for the first four OTP pages, which contain the write-protect bits. If these bits are set, it will prevent the other write-protect bits from being set, thus disabling the write protection mechanism. But this does not prevent the user from programming the other user-programmable OTP pages.

Accessing Private OTP Memory

In order to read or write to the private area of OTP memory, the processor must be operating in Secure Mode and the `OTPSEN` bit within the `SECURE_SYSSWT` register must be set to a value of 1 to enable secured OTP access. For information about Security, Secure Mode and the Secure State Machine, see "Secure State Machine" on page 16-7.

OTP Programming Examples

The following steps are recommended for accessing OTP memory.

1. Initialize OTP array by calling `bfrom_OtpCommand()`.
2. Perform OTP read or write access by calling `bfrom_OtpRead()` or `bfrom_OtpWrite()`.
3. Call `bfrom_OtpCommand()` with `OTP_CLOSE` parameter to re-initialize the OTP controller when OTP read/write access is complete.

4. Initialize OTP array by calling `bfrom_OtpCommand()` for next OTP access.
5. Repeat steps 1–3 for subsequent OTP accesses.

In general, it is recommended that `OTP_CLOSE` be used if sensitive data has been written/read in a secure mode, and the processor is subsequently returned to Open Mode operation. For information about these modes, see Chapter 16, "Security".

To enable access to private OTP memory space while operating in Secure Mode, use the code shown in Listing 4-7.

Listing 4-7. Enable Access to Private OTP

```
/* This code enables access to private OTP via OTPSEN bit */
*pSECURE_SYSSWT = (*pSECURE_SYSSWT) | OTPSEN;
SSYNC();
...
```

To enable JTAG emulation and access to private OTP memory space via `OTPSEN` while operating in Secure Mode, use the code shown in Listing 4-8.

Listing 4-8. Enable Access to Private OTP and Enable JTAG Emulation

```
/* This code enables JTAG and enables access to private OTP
via OTPSEN bit */
*pSECURE_SYSSWT = (*pSECURE_SYSSWT & (~EMUDABL)) | OTPSEN;
SSYNC();
...
```

To read pages 0x4 through 0xDF in public OTP memory space and print results to the VisualDSP++ console, use the code shown in Listing 4-9.

Listing 4-9. Read Pages 0x4 Through 0xDF and Print Results

```
#include <blackfin.h>
#include <bfrom.h>
u32 return_code, i;
u64 value;
//  Initialize OTP timing parameter
//  Proper timing for OTP read access
const u32 OTP_init_value = 0x00001485;
return_code = bfrom_OtpCommand( OTP_INIT, OTP_init_value);
...
for (i= 0x004; <0x0xE0; i++)
   {
return_code = bfrom_OtpRead(i, OTP_LOWER_HALF, &value);
printf("page: 0x%03xL, Content ECC: 0x%01611x, returncode:
       0x%03x \n", i, value, return_code);
return_code = bfrom_OtpRead(i, OTP_UPPER_HALF, &value);
printf("page: 0x%03xH, Content ECC: 0x%01611x, returncode:
       0x%03x \n", i, value, return_code);
    }
```

To write and lock a single OTP page and return the results to the VisualDSP++ console via `printf`, use the code shown in Listing 4-10.

Listing 4-10. Perform OTP Write to a Single Page

```
#include <blackfin.h>
#include <bfrom.h>
u64 value;
u32 return_code;
// Initialize OTP timing parameter
// Proper timing for SCLK = 80 MHz
const u32 OTP_init_value = 0x0A548850;
return_code = bfrom_OtpCommand( OTP_INIT, OTP_init_value);
return_code = bfrom_OtpWrite(0x01C, OTP_LOWER_HALF |
```

```
        OTP_CHECK_FOR_PREV_WRITE, &testdata);
printf("WRITE page: 0x%03xL, Content ECC: 0x%01611x,
       returncode: 0x%03x \n", 0x1C, testdata, return_code);
return_code = bfrom_OtpWrite(0x01C, OTP_UPPER_HALF |
        OTP_CHECK_FOR_PREV_WRITE | OTP_LOCK, &testdata);
printf("WRITE page: 0x%03xH, Content ECC: 0x%01611x,
       returncode: 0x%03x \n", 0x1C, testdata, return_code);
```

Note that locking a page will lock the full 128-bit page, while the previous examples perform OTP access on a 64-bit half-page granularity. This is the finest level of granularity that is allowed due to the OTP error correction implementation. The page lock should occur only after both the lower and upper portion of the page have been written. Note that the page lock operation is performed on the second and final access to the page in the code in Listing 4-10.

The programmer may wish to lock specific OTP pages in a separate access after writing data values is already complete. OTP pages are typically locked to protect them from being overwritten or to prevent inadvertent or malicious tampering. This can be done using the code shown in Listing 4-11.

Listing 4-11. Perform Pure Page Lock without Writing Data

```
#include <blackfin.h>
#include <bfrom.h>
u64 value;
u32 return_code;
// Initialize OTP timing parameter
// Proper timing for SCLK = 80 MHz
const u32 OTP_init_value = 0x0A548850;
return_code = bfrom_OtpCommand( OTP_INIT, OTP_init_value);
return_code = bfrom_OtpWrite(0x01C, OTP_LOCK, NULL);
```

The code shown in Listing 4-12 can be used to read the chip ID code.

Listing 4-12. Read Unique Chip ID

```
#include <bfrom.h>
#include <stdio.h>
#include <cdefBF526.h>
#include <ccblkfn.h> // contains intrinsics for Blackfin
                     // assembler commands
void main()
{
   u32 return_code;      // 32-bit element to hold return code
   u64 idupper, idlower; // Two 64-bit elements to hold the
                // upper & lower halves of the unique chip id
   // Code to read the unique chip ID
   return_code = bfrom_OtpRead(0x4, OTP_LOWER_HALF, &idlower);
   printf("page: 0x%03xL, Content ECC: 0x%016llx, returncode:
          0x%03x\n", 0x4, idlower, return_code);
   return_code = bfrom_OtpRead(0x4, OTP_UPPER_HALF, &idupper);
   printf("page: 0x%03xH, Content ECC: 0x%016llx, returncode:
          0x%03x\n", 0x4, idupper, return_code);
      return;
}
```

5 SYSTEM INTERRUPTS

This chapter discusses the system interrupt controller (SIC). While this chapter does refer to features of the core event controller (CEC), it does not cover all aspects of it. Please refer to the *Blackfin Processor Programming Reference* for more information on the CEC.

Specific Information for the ADSP-BF52x

For details regarding the number of system interrupts for the ADSP-BF52x product, please refer to the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet*.

To determine how each of the system interrupts is multiplexed with other functional pins, refer to Table 9-2 on page 9-5 through Table 9-5 on page 9-9 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for each DMA, refer to Appendix A, "System MMR Assignments".

System interrupt behavior for the ADSP-BF52x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Behavior for the ADSP-BF52x Processor" on page 5-15.

Overview

The processor system has numerous peripherals, which therefore require many supporting interrupts.

Features

The Blackfin architecture provides a two-level interrupt processing scheme:

- The core event controller (CEC) runs in the CCLK clock domain. It interacts closely with the program sequencer and manages the event vector table (EVT). The CEC processes not only core-related interrupts such as exceptions, core errors, and emulation events; it also supports software interrupts.
- The system interrupt controller (SIC) runs in the SCLK clock domain. It masks, groups, and prioritizes interrupt requests signalled by on-chip or off-chip peripherals and forwards them to the CEC.

Description of Operation

The following sections describe the operation of the system interrupts.

Events and Sequencing

The processor employs a two-level event control mechanism. The processor SIC works with the CEC to prioritize and control all system interrupts. The SIC provides mapping between the many peripheral interrupt sources and the prioritized general-purpose interrupt inputs of the core. This mapping is programmable, and individual interrupt sources can be masked in the SIC.

The CEC of the processor manages five types of activities or events:

- Emulation
- Reset
- Nonmaskable interrupts (NMI)

- Exceptions
- Interrupts

Note the word *event* describes all five types of activities. The CEC manages fifteen different events in all: emulation, reset, NMI, exception, and eleven interrupts.

An interrupt is an event that changes the normal processor instruction flow and is asynchronous to program flow. In contrast, an exception is a software initiated event whose effects are synchronous to program flow.

The event system is nested and prioritized. Consequently, several service routines may be active at any time, and a low priority event may be pre-empted by one of higher priority.

The CEC supports nine general-purpose interrupts (`IVG7` – `IVG15`) in addition to the dedicated interrupt and exception events that are described in Table 5-1. It is common for applications to reserve the lowest or the two lowest priority interrupts (`IVG14` and `IVG15`) for software interrupts, leaving eight or seven prioritized interrupt inputs (`IVG7` – `IVG13`) for peripheral purposes. Refer to Table 5-1.

Table 5-1. System and Core Event Mapping

Event Source	Core Event Name
Core events	
Emulation (highest priority)	EMU
Reset	RST
NMI	NMI
Exception	EVX
Reserved	–
Hardware error	IVHW
Core timer	IVTMR

Table 5-1. System and Core Event Mapping (Continued)

Event Source	Core Event Name
System interrupts	IVG7–IVG13
Software interrupt 1	IVG14
Software interrupt 2 (lowest priority)	IVG15

System Peripheral Interrupts

To service the rich set of peripherals, the SIC has multiple interrupt request inputs and outputs that go to the CEC. The primary function of the SIC is to mask, group, and prioritize interrupt requests and to forward them to the nine general-purpose interrupt inputs of the CEC (`IVG7`–`IVG15`). Additionally, the SIC controller can enable individual peripheral interrupts to wake up the processor from Idle or power-down state.

The nine general-purpose interrupt inputs (`IVG7`–`IVG15`) of the core event controller have fixed priority. Of this group, the `IVG7` channel has the highest priority and `IVG15` has the lowest priority. Therefore, the interrupt assignment in the `SIC_IAR` registers not only groups peripheral interrupts; it also programs their priority by assigning them to individual IVG channels. However, the relative priority of peripheral interrupts can be set by mapping the peripheral interrupt to the appropriate general-purpose interrupt level in the core. The mapping is controlled by the `SIC_IAR` register settings shown in Figure 5-2 on page 5-11 and the tables in Appendix A, “System MMR Assignments”. If more than one interrupt source is mapped to the same interrupt, they are logically OR’ed, with no hardware prioritization. Software can prioritize the interrupt processing as required for a particular system application.

For general-purpose interrupts with multiple peripheral interrupts assigned to them, take special care to ensure that software correctly processes all pending interrupts sharing that input. Software is responsible for prioritizing the shared interrupts.

The core timer has a dedicated input to the CEC controller. Its interrupt is not routed through the SIC controller and always has higher priority than requests from all peripherals.

The SIC_IMASK register allows software to mask any peripheral interrupt source at the SIC level. This functionality is independent of whether the particular interrupt is enabled at the peripheral itself. At reset, the contents of the SIC_IMASK register are all 0s to mask off all peripheral interrupts. Turning off a system interrupt mask and enabling the particular interrupt is performed by writing a 1 to a bit location in the SIC_IMASK register.

The SIC includes one or more read-only SIC_ISR registers with individual bits which correspond to the interrupt status of one of the peripheral interrupt sources. When the SIC detects the interrupt, the bit is asserted. When the SIC detects that the peripheral interrupt input has been deasserted, the respective bit in the system interrupt status register is cleared. Note for some peripherals, such as general-purpose I/O asynchronous input interrupts, many cycles of latency may pass from the time an interrupt service routine initiates the clearing of the interrupt (usually by writing a system MMR) to the time the SIC senses that the interrupt has been deasserted.

Depending on how interrupt sources map to the general-purpose interrupt inputs of the core, the interrupt service routine may have to interrogate multiple interrupt status bits to determine the source of the interrupt. One of the first instructions executed in an interrupt service routine should read the SIC_ISR register to determine whether more than one of the peripherals sharing the input has asserted its interrupt output. The service routine should fully process all pending, shared interrupts before

executing the RTI, which enables further interrupt generation on that interrupt input.

When an interrupt's service routine is finished, the RTI instruction clears the appropriate bit in the IPEND register. However, the relevant SIC_ISR bit is not cleared unless the service routine clears the mechanism that generated the interrupt.

Many systems need relatively few interrupt-enabled peripherals, allowing each peripheral to map to a unique core priority level. In these designs, the SIC_ISR register will seldom, if ever, need to be interrogated.

The SIC_ISR register is not affected by the state of the SIC_IMASK register and can be read at any time. Writes to the SIC_ISR register have no effect on its contents.

Peripheral DMA channels are mapped in a fixed manner to the peripheral interrupt IDs. However, the assignment between peripherals and DMA channels is freely programmable with the DMA_PERIPHERAL_MAP registers. Table 5-1 on page 5-3 and Table 5-2 on page 5-11 show the default DMA assignment. Once a peripheral has been assigned to any other DMA channel it uses the new DMA channel's interrupt ID regardless of whether DMA is enabled or not. Therefore, clean DMA_PERIPHERAL_MAP management is required even if the DMA is not used. The default setup should be the best choice for all non-DMA applications.

For dynamic power management, any of the peripherals can be configured to wake up the core from its idled state to process the interrupt, simply by enabling the appropriate bit in the SIC_IWR register (refer to Table 5-1 on page 5-3 and Table 5-2 on page 5-11). If a peripheral interrupt source is enabled in SIC_IWR and the core is idled, the interrupt causes the DPMC to initiate the core wakeup sequence in order to process the interrupt. Note this mode of operation may add latency to interrupt processing, depending on the power control state. For further discussion of power modes and the idled state of the core, see the Dynamic Power Management chapter.

The SIC_IWR register has no effect unless the core is idled. By default, all interrupts generate a wakeup request to the core. However, for some applications it may be desirable to disable this function for some peripherals, such as for a SPORT transmit interrupt. The SIC_IWR register can be read from or written to at any time. To prevent spurious or lost interrupt activity, this register should be written to only when all peripheral interrupts are disabled.

The wakeup function is independent of the interrupt mask function. If an interrupt source is enabled in the SIC_IWR but masked off in the SIC_IMASK register, the core wakes up if it is idled, but it does not generate an interrupt.

The peripheral interrupt structure of the processor is flexible. Upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core by default, as shown in Table 5-2 on page 5-11.

An interrupt service routine that supports multiple interrupt sources must interrogate the appropriate system memory mapped registers (MMRs) to determine which peripheral generated the interrupt.

Programming Model

The programming model for the system interrupts is described in the following sections.

System Interrupt Initialization

If the default peripheral-to-IVG assignments shown in Table 5-1 on page 5-3 and Table 5-2 on page 5-11 are acceptable, then interrupt initialization involves only:

- Initialization of the core event vector table (EVT) vector address entries
- Initialization of the IMASK register
- Unmasking the specific peripheral interrupts that the system requires in the SIC_IMASK register

System Interrupt Processing Summary

Referring to Figure 5-1 on page 5-10, note when an interrupt (interrupt A) is generated by an interrupt-enabled peripheral:

1. SIC_ISR logs the request and keeps track of system interrupts that are asserted but not yet serviced (that is, an interrupt service routine hasn't yet cleared the interrupt).
2. SIC_IWR checks to see if it should wake up the core from an idled state based on this interrupt request.
3. SIC_IMASK masks off or enables interrupts from peripherals at the system level. If interrupt A is not masked, the request proceeds to Step 4.
4. The SIC_IAR registers, which map the peripheral interrupts to a smaller set of general-purpose core interrupts (IVG7 - IVG15), determine the core priority of interrupt A.
5. ILAT adds interrupt A to its log of interrupts latched by the core but not yet actively being serviced.

6. IMASK masks off or enables events of different core priorities. If the IVGx event corresponding to interrupt A is not masked, the process proceeds to Step 7.
7. The event vector table (EVT) is accessed to look up the appropriate vector for interrupt A's interrupt service routine (ISR).
8. When the event vector for interrupt A has entered the core pipeline, the appropriate IPEND bit is set, which clears the respective ILAT bit. Thus, IPEND tracks all pending interrupts, as well as those being presently serviced.
9. When the interrupt service routine (ISR) for interrupt A has been executed, the RTI instruction clears the appropriate IPEND bit. However, the relevant SIC_ISR bit is not cleared unless the interrupt service routine clears the mechanism that generated interrupt A, or if the process of servicing the interrupt clears this bit.

It should be noted that emulation, reset, NMI, and exception events, as well as hardware error (IVHW) and core timer (IVTMR) interrupt requests, enter the interrupt processing chain at the ILAT level and are not affected by the system-level interrupt registers (SIC_IWR, SIC_ISR, SIC_IMASK, SIC_IAR).

If multiple interrupt sources share a single core interrupt, then the interrupt service routine (ISR) must identify the peripheral that generated the

interrupt. The ISR may then need to interrogate the peripheral to determine the appropriate action to take.

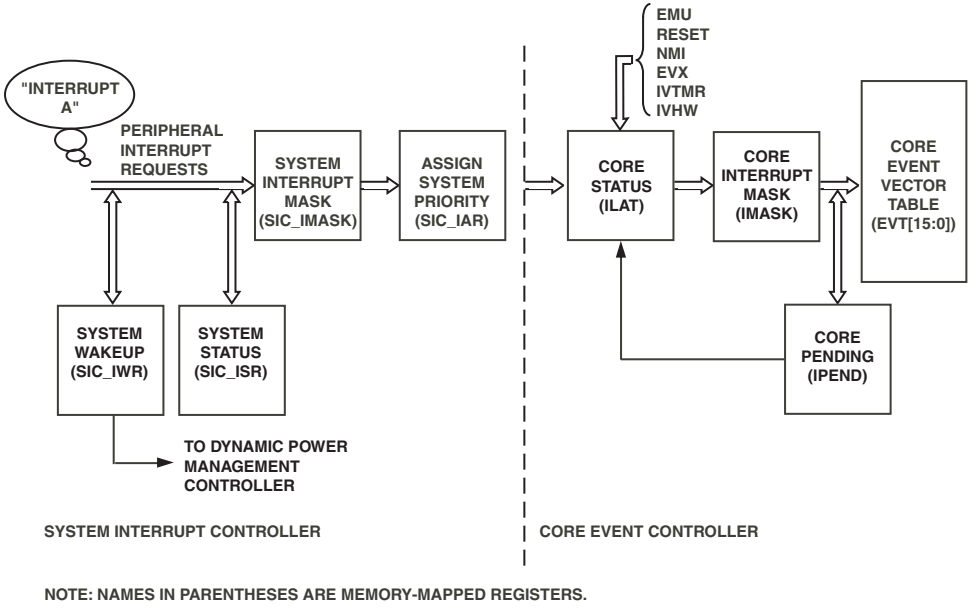


Figure 5-1. Interrupt Processing Block Diagram

System Interrupt Controller Registers

The SIC registers are described in the following sections.

These registers can be read from or written to at any time in supervisor mode. It is advisable, however, to configure them in the reset interrupt service routine before enabling interrupts. To prevent spurious or lost interrupt activity, these registers should be written to only when all peripheral interrupts are disabled.

System Interrupt Assignment (SIC_IAR) Register

The SIC_IAR register maps each peripheral interrupt ID to a corresponding IVG priority level. This is accomplished with 4-bit groupings that translate to IVG levels as shown in Table 5-2 and Figure 5-2 on page 5-11. In other words, Table 5-2 defines the value to write in a 4-bit field within SIC_IAR in order to configure a peripheral interrupt ID for a particular IVG priority. Refer to Table 5-1 on page 5-3 for information on SIC_IAR mappings for this specific processor.

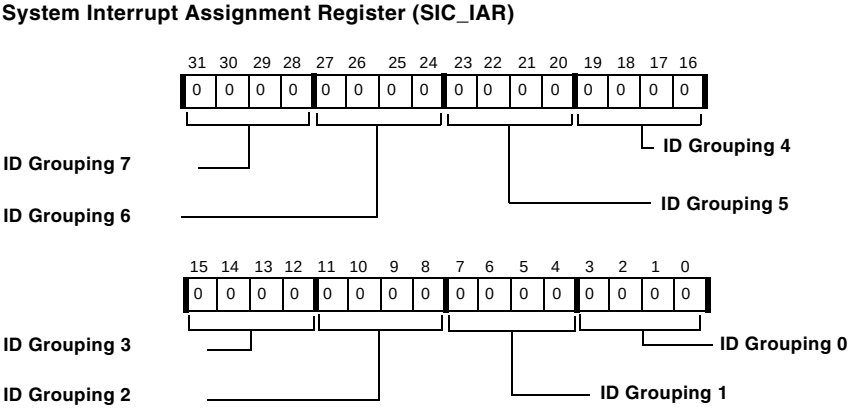

System Interrupt Assignment Register (SIC_IAR)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ID Grouping 7
ID Grouping 6
ID Grouping 4
ID Grouping 5
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ID Grouping 3
ID Grouping 2
ID Grouping 0
ID Grouping 1

Figure 5-2. System Interrupt Assignment Register

Table 5-2. IVG Select Definitions

General-purpose Interrupt	Value in SIC_IAR
IVG7	0
IVG8	1
IVG9	2
IVG10	3
IVG11	4
IVG12	5

Table 5-2. IVG Select Definitions (Continued)

General-purpose Interrupt	Value in SIC_IAR
IVG13	6
IVG14	7
IVG15	8

System Interrupt Mask (SIC_IMASK) Register

The SIC_IMASK register masks or enables peripheral interrupts at the system level. A "0" in a bit position masks off (disables) interrupts for that particular peripheral interrupt ID. A "1" enables interrupts for that interrupt ID. Refer to Table 5-1 on page 5-3 and Table 5-2 on page 5-11 for information on how peripheral interrupt IDs are mapped to the SIC_IMASK register(s) for this particular processor.

System Interrupt Status (SIC_ISR) Register

The SIC_ISR register keeps track of system interrupts that are asserted but not yet serviced. A "0" in a bit position indicates that a particular interrupt is deasserted. A "1" indicates that it is asserted. Refer to Table 5-1 on page 5-3 and Table 5-2 on page 5-11 for information on how peripheral interrupt IDs are mapped to the SIC_ISR register(s) for this particular processor.

System Interrupt Wakeup-Enable (SIC_IWR) Register

The SIC_IWR register allows an interrupt request to wake up the processor core from an idled state. A "0" in a bit position indicates that a particular peripheral interrupt ID is not configured to wake the core (upon assertion of the interrupt request). A "1" indicates that it is configured to do so. Refer to Table 5-1 on page 5-3 and Table 5-2 on page 5-11 for informa-

tion on how peripheral interrupt IDs are mapped to the SIC_IWR register(s) for this particular processor.

Programming Examples

The following section provides an example for servicing interrupt requests.

Clearing Interrupt Requests

When the processor services a core event it automatically clears the requesting bit in the ILAT register and no further action is required by the interrupt service routine. It is important to understand that the SIC controller does not provide any interrupt acknowledgment feedback mechanism from the CEC controller back to the peripherals. Although the ILAT bits clear in the same way when a peripheral interrupt is serviced, the signalling peripheral does not release its level-sensitive request until it is explicitly instructed by software. If however, the peripheral keeps requesting, the respective ILAT bit is set again immediately and the service routine is invoked again as soon as its first run terminates by an RTI instruction.

Every software routine that services peripheral interrupts must clear the signalling interrupt request in the respective peripheral. The individual peripherals provide customized mechanisms for how to clear interrupt requests. Receive interrupts, for example, are cleared when received data is read from the respective buffers. Transmit requests typically clear when software (or DMA) writes new data into the transmit buffers. These implicit acknowledge mechanisms avoid the need for cycle-consuming software handshakes in streaming interfaces. Other peripherals such as timers, GPIOs, and error requests require explicit acknowledge instructions, which are typically performed by efficient W1C (write-1-to-clear) operations.

Listing 5-1 shows a representative example of how a GPIO interrupt request might be serviced.

Listing 5-1. Servicing GPIO Interrupt Request

```
#include <defBF527.h>
/*ADSP-BF527 product is used as an example*/
.section program;
_portg_a_isr:
  /* push used registers */
  [--sp] = (r7:7, p5:5);
  /* clear interrupt request on GPIO pin PG2 */
  /* no matter whether used A or B channel */
  p5.l = lo(PORTGIO_CLEAR);
  p5.h = hi(PORTGIO_CLEAR);
  r7 = PG2;
  w[p5] = r7;

  /*  place user code here */

  /* sync system, pop registers and exit */
  ssync;
  (r7:7, p5:5) = [sp++];
  rti;
_portg_a_isr.end:
```

The W1C instruction shown in this example may require several SCLK cycles to complete, depending on system load and instruction history. The program sequencer does not wait until the instruction completes and continues program execution immediately. The SSYNC instruction ensures that the W1C command indeed cleared the request in the GPIO peripheral before the RTI instruction executes. However, the SSYNC instruction does not guarantee that the release of interrupt request has also been recognized by the CEC controller, which may require a few more CCLK cycles depending on the CCLK-to-SCLK ratio. In service routines consisting of a few

instructions only, two SSYNC instructions are recommended between the clear command and the RTI instruction. However, one SSYNC instruction is typically sufficient if the clear command performs in the very beginning of the service routine, or the SSYNC instruction is followed by another set of instructions before the service routine returns. Commonly, a pop-multiple instruction is used for this purpose as shown in Listing 5-1.

The level-sensitive nature of peripheral interrupts enables more than one of them to share the same IVG channel and therefore the same interrupt priority. This is programmable using the assignment registers. Then a common service routine typically interrogates the SIC_ISR register to determine the signalling interrupt source. If multiple peripherals are requesting interrupts at the same time, it is up to the service routine to either service all requests in a single pass or to service them one by one. If only one request is serviced and the respective request is cleared by software before the RTI instruction executes, the same service routine is invoked another time because the second request is still pending. While the first approach may require fewer cycles to service both requests, the second approach enables higher priority requests to be serviced more quickly in a non-nested interrupt system setup.

Unique Behavior for the ADSP-BF52x Processor

Interfaces

Figure 5-3 on page 5-16 and Figure 5-4 on page 5-17 provide an overview of how the individual peripheral interrupt request lines connect to the

SIC. These figures show how the eight SIC_IAR registers control the assignment to the nine available peripheral request inputs of the CEC.

The memory-mapped ILAT, IMASK, and IPEND registers are part of the CEC controller.

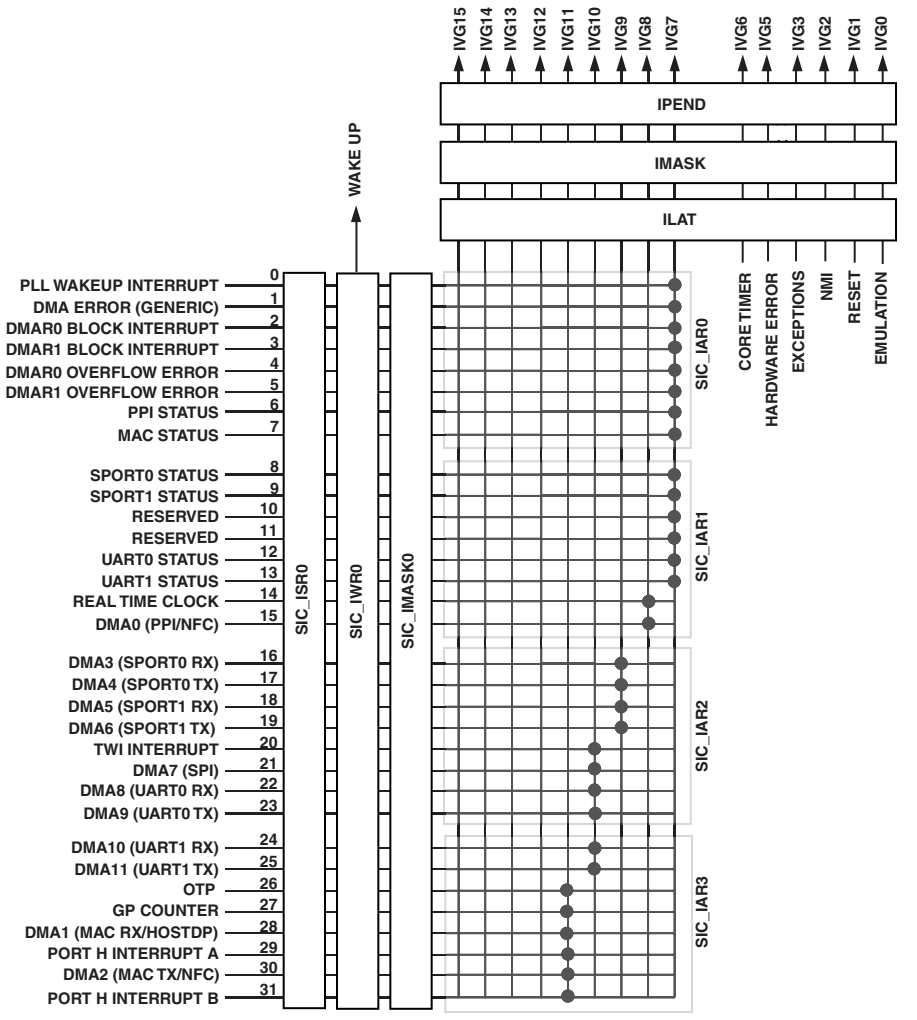


Figure 5-3. Interrupt Routing Overview (Part 1)

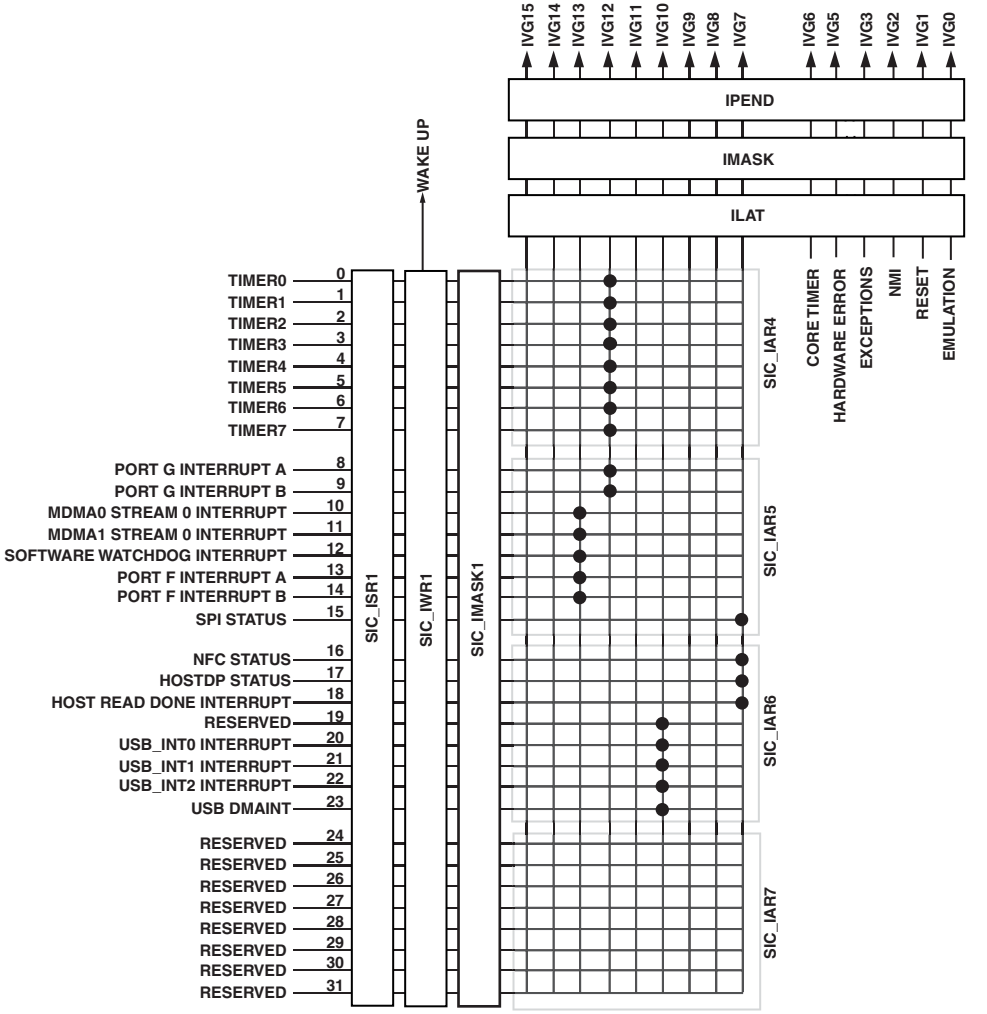


Figure 5-4. Interrupt Routing Overview (Part 2)

System Peripheral Interrupts

The MAC interrupt requests shown in Figure 5-3 on page 5-16 are available only on ADSP-BF527 parts. However, for code compatibility, all of the default assignments for the ADSP-BF52x processors are the same.

Table 5-3 and Table 5-4 on page 5-20 show the peripheral interrupt events, the default mapping of each event, the peripheral interrupt ID used in the system interrupt assignment registers (SIC_IAR), and the core interrupt ID.

Note that the system interrupt to core event mappings shown are the default values at reset and can be changed by software. Where there is more than one DMA interrupt source for a given interrupt ID number, the default DMA source mapping is listed first in parentheses.

The peripheral interrupt structure of the processor is flexible. Upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core by default, as shown in Table 5-3 on page 5-18 and Table 5-4 on page 5-20.

An interrupt service routine that supports multiple interrupt sources must interrogate the appropriate system memory mapped registers (MMRs) to determine which peripheral generated the interrupt.

Table 5-3. Peripheral Interrupt Events (Part 1)

Peripheral ID#	Bit Position	SIC_ISR0, SIC_IMASK0, SIC_IWR0	SIC_IAR3–0	Default Mapping
31	Bit 31	Port H interrupt B	SIC_IAR3[31:28]	IVG11
30	Bit 30	DMA2 (Ethernet MAC TX/NFC)	SIC_IAR3[27:24]	IVG11
29	Bit 29	Port H interrupt A	SIC_IAR3[23:20]	IVG11
28	Bit 28	DMA1 (Ethernet MAC RX/HOSTDP)	SIC_IAR3[19:16]	IVG11
27	Bit 27	GP Counter	SIC_IAR3[15:12]	IVG11
26	Bit 26	OTP Memory	SIC_IAR3[11:8]	IVG11

Table 5-3. Peripheral Interrupt Events (Part 1) (Continued)

Peripheral ID#	Bit Position	SIC_ISR0, SIC_IMASK0, SIC_IWR0	SIC_IAR3–0	Default Mapping
25	Bit 25	DMA11 (UART1 TX)	SIC_IAR3[7:4]	IVG10
24	Bit 24	DMA10 (UART1 RX)	SIC_IAR3[3:0]	IVG10
23	Bit 23	DMA9 (UART0 TX)	SIC_IAR2[31:28]	IVG10
22	Bit 22	DMA8 (UART0 RX)	SIC_IAR2[27:24]	IVG10
21	Bit 21	DMA7 (SPI RX/TX)	SIC_IAR2[23:20]	IVG10
20	Bit 20	TWI	SIC_IAR2[19:16]	IVG10
19	Bit 19	DMA6 (SPORT1 TX)	SIC_IAR2[15:12]	IVG9
18	Bit 18	DMA5 (SPORT1 RX)	SIC_IAR2[11:8]	IVG9
17	Bit 17	DMA4 (SPORT0 TX)	SIC_IAR2[7:4]	IVG9
16	Bit 16	DMA3 (SPORT0 RX)	SIC_IAR2[3:0]	IVG9
15	Bit 15	DMA0 (PPI/NFC)	SIC_IAR1[31:28]	IVG8
14	Bit 14	Real-time clock	SIC_IAR1[27:24]	IVG8
13	Bit 13	UART1 status	SIC_IAR1[23:20]	IVG7
12	Bit 12	UART0 status	SIC_IAR1[19:16]	IVG7
11	Bit 11	Reserved	SIC_IAR1[15:12]	IVG7
10	Bit 10	Reserved	SIC_IAR1[11:8]	IVG7
9	Bit 9	SPORT1 status	SIC_IAR1[7:4]	IVG7
8	Bit 8	SPORT0 status	SIC_IAR1[3:0]	IVG7
7	Bit 7	Ethernet MAC status	SIC_IAR0[31:28]	IVG7
6	Bit 6	PPI error	SIC_IAR0[27:24]	IVG7
5	Bit 5	DMAR1 overflow error	SIC_IAR0[23:20]	IVG7
4	Bit 4	DMAR0 overflow error	SIC_IAR0[19:16]	IVG7
3	Bit 3	DMAR1 block interrupt	SIC_IAR0[15:12]	IVG7
2	Bit 2	DMAR0 block interrupt	SIC_IAR0[11:8]	IVG7

Table 5-3. Peripheral Interrupt Events (Part 1) (Continued)

Peripheral ID#	Bit Position	SIC_ISR0, SIC_IMASK0, SIC_IWR0	SIC_IAR3–0	Default Mapping
1	Bit 1	DMA Error (generic)	SIC_IAR0[7:4]	IVG7
0	Bit 0	PLL Wakeup	SIC_IAR0[3:0]	IVG7

Table 5-4. Peripheral Interrupt Events (Part 2)

Peripheral ID#	Bit Position	SIC_ISR1, SIC_IMASK1, SIC_IWR1	SIC_IAR7–4	Default Mapping
63	Bit 31	Reserved	SIC_IAR7[31:28]	IVG13
62	Bit 30	Reserved	SIC_IAR7[27:24]	IVG13
61	Bit 29	Reserved	SIC_IAR7[23:20]	IVG13
60	Bit 28	Reserved	SIC_IAR7[19:16]	IVG12
59	Bit 27	Reserved	SIC_IAR7[15:12]	IVG12
58	Bit 26	Reserved	SIC_IAR7[11:8]	IVG12
57	Bit 25	Reserved	SIC_IAR7[7:4]	IVG12
56	Bit 24	Reserved	SIC_IAR7[3:0]	IVG12
55	Bit 23	USB_DMAINT	SIC_IAR6[31:28]	IVG10
54	Bit 22	USB_INT2	SIC_IAR6[27:24]	IVG10
53	Bit 21	USB_INT1	SIC_IAR6[23:20]	IVG10
52	Bit 20	USB_INT0	SIC_IAR6[19:16]	IVG10
51	Bit 19	Reserved	SIC_IAR6[15:12]	IVG10
50	Bit 18	Host read done	SIC_IAR6[11:8]	IVG7
49	Bit 17	HOSTDP status	SIC_IAR6[7:4]	IVG7
48	Bit 16	NFC status	SIC_IAR6[3:0]	IVG7
47	Bit 15	SPI status	SIC_IAR5[31:28]	IVG7
46	Bit 14	Port F interrupt B	SIC_IAR5[27:24]	IVG13
45	Bit 13	Port F interrupt A	SIC_IAR5[23:20]	IVG13
44	Bit 12	Watchdog timer	SIC_IAR5[19:16]	IVG13

Table 5-4. Peripheral Interrupt Events (Part 2) (Continued)

Peripheral ID#	Bit Position	SIC_ISR1, SIC_IMASK1, SIC_IWR1	SIC_IAR7–4	Default Mapping
43	Bit 11	MDMA1	SIC_IAR5[15:12]	IVG13
42	Bit 10	MDMA0	SIC_IAR5[11:8]	IVG13
41	Bit 9	Port G interrupt B	SIC_IAR5[7:4]	IVG12
40	Bit 8	Port G interrupt A	SIC_IAR5[3:0]	IVG12
39	Bit 7	Timer 7	SIC_IAR4[31:28]	IVG12
38	Bit 6	Timer 6	SIC_IAR4[27:24]	IVG12
37	Bit 5	Timer 5	SIC_IAR4[23:20]	IVG12
36	Bit 4	Timer 4	SIC_IAR4[19:16]	IVG12
35	Bit 3	Timer 3	SIC_IAR4[15:12]	IVG12
34	Bit 2	Timer 2	SIC_IAR4[11:8]	IVG12
33	Bit 1	Timer 1	SIC_IAR4[7:4]	IVG12
32	Bit 0	Timer 0	SIC_IAR4[3:0]	IVG12

6 DIRECT MEMORY ACCESS

This chapter describes the direct memory access (DMA) controller. Following an overview and list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

This chapter describes the features common to all the DMA channels, as well as how DMA operations are set up. For specific peripheral features, see the appropriate peripheral chapter for additional information. Performance and bus arbitration for DMA operations can be found in Chapter 2, "Chip Bus Hierarchy".

Specific Information for the ADSP-BF52x

For details regarding the number of DMA controllers for the ADSP-BF52x product, please refer to the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet*.

For DMA interrupt vector assignments, refer to Table 5-3 on page 5-18 in Chapter 5, "System Interrupts".

To determine how each of the DMAs is multiplexed with other functional pins, refer to Table 9-2 on page 9-5 through Table 9-5 on page 9-9 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for each DMA, refer to Appendix A, "System MMR Assignments".

DMA controller behavior for the ADSP-BF52x that differs from the general information in this chapter can be found in the section "Unique Behavior for the ADSP-BF52x Processor" on page 6-105

Overview and Features

The processor uses DMA to transfer data between memory spaces or between a memory space and a peripheral. The processor can specify data transfer operations and return to normal processing while the fully integrated DMA controller carries out the data transfers independent of processor activity.

The DMA controller can perform several types of data transfers:

- Peripheral DMA transfers data between memory and on-chip peripherals.
- Memory DMA (MDMA) transfers data between memory and memory. The processor has two MDMA modules, each consisting of independent memory read and memory write channels.
- Handshaking memory DMA (HMDMA) transfers data between off-chip peripherals and memory. This enhancement of the MDMA channels enables external hardware to control the timing of individual data transfers or block transfers.

The HMDMA feature is not available for all products. Refer to "Unique Behavior for the ADSP-BF52x Processor" on page 6-105 to determine whether it applies to this product.

All DMAs can transport data to and from on-chip and off-chip memories, including L1 and SDRAM. The L1 scratchpad memory cannot be accessed by DMA.

SDRAM and SRAM are not available on all products. Refer to "Unique Behavior for the ADSP-BF52x Processor" on page 6-105 to determine whether it applies to this product.

DMA transfers on the processor can be descriptor-based or register-based.

Register-based DMA allows the processor to directly program DMA control registers to initiate a DMA transfer. On completion, the control registers may be automatically updated with their original setup values for continuous transfer, if needed.

Descriptor-based DMA transfers require a set of parameters stored within memory to initiate a DMA sequence. This sort of transfer allows the chaining together of multiple DMA sequences. In descriptor-based DMA operations, a DMA channel can be programmed to automatically set up and start another DMA transfer after the current sequence completes.

Examples of DMA styles supported by flex descriptors include:

- A single linear buffer that stops on completion (`FLOW` = stop mode)
- A linear buffer with byte strides of any integer value, including negative values (`DMAx_X_MODIFY` register)
- A circular, auto-refreshing buffer that interrupts on each full buffer
- A similar buffer that interrupts on fractional buffers (for example, ½, ¼) (2-D DMA)
- 1-D DMA, using a set of identical ping-pong buffers defined by a linked ring of 3-word descriptors, each containing a link pointer and a 32-bit address

- 1-D DMA, using a linked list of 5-word descriptors containing a link pointer, a 32-bit address, the buffer length, and a configuration
- 2-D DMA, using an array of 1-word descriptors, specifying only the base DMA address within a common data page
- 2-D DMA, using a linked list of 9-word descriptors specifying everything

DMA Controller Overview

A block diagram of the DMA controller can be found in the "Unique Behavior for the ADSP-BF52x Processor" on page 6-105.

External Interfaces

The DMA does not connect external memories and devices directly. Rather, data is passed through the EBIU port. Any kind of device that is supported by the EBIU can also be accessed by peripheral DMA or memory DMA operation. This is typically flash memory, SRAM, SDRAM, FIFOs, or memory-mapped peripheral devices.

For products with handshaking MDMA (HMDMA), the operation is supported by two MDMA request input pins, `DMAR0` and `DMAR1`. The `DMAR0` pin controls transfer timing on the `MDMA0` destination channel. The `DMAR1` pin controls the destination channel of `MDMA1`. With these pins, external FIFO devices, ADC or DAC converters, or other streaming or block-processing devices can use the MDMA channels to exchange their data or data buffers with the Blackfin processor memory.

Internal Interfaces

Figure 2-1 on page 2-3 shows the dedicated DMA buses used by the DMA controller to interconnect L1 memory, the on-chip peripherals, and the EBIU port.

The 16-bit DMA core bus (DCB) connects the DMA controller to a dedicated port of L1 memory. L1 memory has dedicated DMA ports featuring special DMA buffers to decouple DMA operation. See the Blackfin Processor Programming Reference for a description of the L1 memory architecture. The DCB bus operates at core clock (CCLK) frequency. It is the DMA controller's responsibility to translate DCB transfers to the system clock (SCLK) domain.

The 16-bit DMA access bus (DAB) connects the DMA controller to the on-chip peripherals. This bus operates at SCLK frequency.

The 16-bit DMA external bus (DEB) connects the DMA controller to the EBIU port. This bus is used for all peripheral and memory DMA transfers to and from external memories and devices. It operates at SCLK frequency.

Transferred data can be 8-, 16-, or 32-bits wide. The DMA controller, however, connects only to 16-bit buses.

Memory DMA can pass data every SCLK cycle between L1 memory and the EBIU. Transfers from L1 memory to L1 memory require two cycles, as the DCB bus is used for both source and destination transfers. Similarly, transfers between two off-chip devices require EBIU and DEB resources twice. Peripheral DMA transfers can be performed every other SCLK cycle.

For more details on DMA performance see "DMA Performance" on page 6-42.

Peripheral DMA

The DMA controller features 12 channels that perform transfers between peripherals and on-chip or off-chip memories. The user has full control over the mapping of DMA channels and peripherals. The default DMA channel priority and mapping, shown in Table 6-7 on page 6-106, can be changed by altering the 4-bit PMAP field in the DMAx_PERIPHERAL_MAP registers for the peripheral DMA channels.

The default configuration should suffice in most cases, but there are some cases where remapping the assignment can be helpful because of the DMA channel priorities. When competing for any of the system buses, DMA0 has higher priority than DMA1, and so on. DMA11 has the lowest priority of the peripheral DMA channels.

A 1:1 mapping should exist between DMA channels and peripherals. The user is responsible for ensuring that multiple DMA channels are not mapped to the same peripheral and that multiple peripherals are not mapped to the same DMA port. If multiple channels are mapped to the same peripheral, only one channel is connected (the lowest priority channel). If a nonexistent peripheral (for example, 0xF in the PMAP field) is mapped to a channel, that channel is disabled—DMA requests are ignored, and no DMA grants are issued. The DMA requests are also not forwarded from the peripheral to the interrupt controller.

All peripheral DMA channels work completely independently from each other. The transfer timing is controlled by the mapped peripheral.

Every DMA channel features its own 4-deep FIFO that decouples DAB activity from DCB and DEB availability. DMA interrupt and descriptor fetch timing is aligned with the memory side (DCB/DEB side) of the FIFO. The user does, however, have an option to align interrupts with the peripheral side (DAB side) of the FIFO for transmit operations. Refer to the SYNC bit in the DMAx_CONFIG register for details.

Memory DMA

This section describes the two pairs of MDMA channels, which provide memory-to-memory DMA transfers among the various memory spaces. These include L1 memory and external synchronous/asynchronous memories.

Each MDMA channel contains a DMA FIFO, an 8-word by 16-bit FIFO block used to transfer data to and from either L1 or the DCB and DEB buses. Typically, it is used to transfer data between external memory and internal memory. It will also support DMA from the boot ROM on the DEB bus. The FIFO can be used to hold DMA data transferred between two L1 memory locations or between two external memory locations.

Each page of MDMA channels consists of:

- A source channel (for reading from memory)
- A destination channel (for writing to memory)

A memory-to-memory transfer always requires both the source and the destination channel to be enabled. Each source/destination channel forms a “stream,” and these two streams are hardwired for DMA priorities 12 through 15.

- Priority 12: MDMA0 destination
- Priority 13: MDMA0 source
- Priority 14: MDMA1 destination
- Priority 15: MDMA1 source

MDMA0 takes precedence over MDMA1, unless round-robin scheduling is used or priorities become urgent, as programmed by the DRQ bit field in the HMDMA_CONTROL register.

It is illegal to program a source channel for memory write or a destination channel for memory read.

The channels support 8-, 16-, and 32-bit memory DMA transfers, but both ends of the MDMA connect to 16-bit buses. Source and destination channels must be programmed to the same word size. In other words, the MDMA transfer does not perform packing or unpacking of data; each read results in one write. Both ends of the MDMA FIFO for a given stream are granted priority at the same time. Each pair shares an 8-word deep 16-bit FIFO. The source DMA engine fills the FIFO, while the destination DMA engine empties it. The FIFO depth allows the burst transfers of the external access bus (EAB) and DMA access bus (DAB) to overlap, significantly improving throughput on block transfers between internal and external memory. Two separate descriptor blocks are required to supply the operating parameters for each MDMA pair, one for the source channel and one for the destination channel.

Because the source and destination DMA engines share a single FIFO buffer, the descriptor blocks must be configured to have the same data size. It is possible to have a different mix of descriptors on both ends as long as the total transfer count is the same.

To start a MDMA transfer operation, the MMRs for the source and destination channels are written, each in a manner similar to peripheral DMA.

The DMAx_CONFIG register for the source channel must be written before the DMAx_CONFIG register for the destination channel.

Handshaked Memory DMA (HMDMA) Mode

This feature is not available for all products. Refer to the "Unique Behavior for the ADSP-BF52x Processor" on page 6-105 to determine whether it applies to this product.

Handshaked operation applies only to memory DMA channels.

Normally, memory DMA transfers are performed at maximum speed. Once started, data is transferred in a continuous manner until either the data count expires or the MDMA is stopped. In handshake mode, the MDMA does not transfer data automatically when enabled; it waits for an external trigger on the MDMA request input signals. The DMAR0 input is associated with MDMA0 and the DMAR1 input with MDMA1. Once a trigger event is detected, a programmable portion of data is transferred and then the MDMA stalls again and waits for the next trigger.

Handshake operation is not only useful for controlling the timing of memory-to-memory transfers, it also enables the MDMA to operate with asynchronous FIFO-style devices connected to the EBIU port. The Blackfin processor acknowledges a DMA request by a proper number of read or write operations. It is up to the device connected to any of the AMSx strobes to deassert or pulse the request signal and to decrement the number of pending requests accordingly.

Depending on HMDMA operating mode, an external DMA request may trigger individual data word transfers or block transfers. A block can consist of up to 65535 data words. For best throughput, DMA requests can be pipelined. The HMDMA controllers feature a request counter to decouple request timing from the data transfers.

See "Handshaked Memory DMA Operation" on page 6-107 for a functional description.

Modes of Operation

The following sections describe the DMA operation.

Register-Based DMA Operation

Register-based DMA is the traditional kind of DMA operation. Software configures the source or destination address and the length of the data to be transferred to memory-mapped registers and then starts DMA operation.

For basic operation, the software performs these steps:

- Write the source or destination address to the 32-bit DMAx_START_ADDR register.
- Write the number of data words to be transferred to the 16-bit DMAx_X_COUNT register.
- Write the address modifier to the 16-bit DMAx_X_MODIFY register. This is the two's-complement value added to the address pointer after every transfer. This value must always be initialized as there is no default value. Typically, this register is set to 0x0004 for 32-bit DMA transfers, to 0x0002 for 16-bit transfers, and to 0x0001 for byte transfers.
- Write the operation mode to the DMAx_CONFIG register. These bits in particular need to be changed as needed:
    - The DMAEN bit enables the DMA channel.
    - The WNR bit controls the DMA direction. DMAs that read from memory (peripheral transmit DMAs and source channel MDMAs) keep this bit cleared. Peripheral receive DMAs and destination channel MDMAs set this bit because they write to memory.

- The WDSIZE bit controls the data word width for the transfer. It can be 8-, 16-, or 32-bits wide.
- The DI_EN bit enables an interrupt when the DMA operation has finished.
- Set the FLOW field to 0x0 for stop mode or 0x1 for autobuffer mode.

Once the DMAEN bit is set, the DMA channel starts its operation. While running, the DMAx_CURR_ADDR and the DMAx_CURR_X_COUNT registers can be monitored to determine the current progress of the DMA operation. However they should not be used to synchronize software and hardware.

The DMAx_IRQ_STATUS register signals whether the DMA has finished (DMA_DONE bit), whether a DMA error has occurred (DMA_ERR bit), and whether the DMA is currently running (DMA_RUN bit). The DMA_DONE and the DMA_ERR bits also function as interrupt latch bits. They must be cleared by write-one-to-clear (W1C) operations by the interrupt service routine.

Stop Mode

In stop mode, the DMA operation is executed only once. When started, the DMA channel transfers the desired number of data words and stops itself when the transfer is complete. If the DMA channel is no longer used, software should clear the DMAEN enable bit to disable the otherwise paused channel. Stop mode is entered if the FLOW bit field in the DMA channel's DMAx_CONFIG register is 0. The NDSIZE field must always be 0 in this mode.

For receive (memory write) operation, the DMA_RUN bit functions almost the same as the inverted DMA_DONE bit. For transmit (memory read) operation, however, the two bits have different timing. Refer to the description of the SYNC bit in the DMAx_CONFIG register for details.

Autobuffer Mode

In autobuffer mode, the DMA operates repeatedly in a circular manner. If all data words have been transferred, the address pointer DMAx_CURR_ADDR is reloaded automatically by the DMAx_START_ADDR value. An interrupt may also be generated.

Autobuffer mode is entered if the FLOW field in the DMAx_CONFIG register is 1. The NDSIZE bit must be 0 in autobuffer mode.

Two-Dimensional DMA Operation

Register-based and descriptor-based DMA can operate in one-dimensional mode or two-dimensional mode.

In two-dimensional (2-D) mode, the DMAx_X_COUNT register is accompanied by the DMAx_Y_COUNT register, supporting arbitrary row and column sizes up to 64K × 64K elements, as well as arbitrary DMAx_X_MODIFY and DMAx_Y_MODIFY values up to ±32K bytes. Furthermore, DMAx_Y_MODIFY can be negative, allowing implementation of interleaved datastreams. The DMAx_X_COUNT and DMAx_Y_COUNT values specify the row and column sizes, where DMAx_X_COUNT must be 2 or greater.

The start address and modify values are in bytes, and they must be aligned to a multiple of the DMA transfer word size (WDSIZE[1:0] in DMAx_CONFIG). Misalignment causes a DMA error.

The DMAx_X_MODIFY value is the byte-address increment that is applied after each transfer that decrements the DMAx_CURR_X_COUNT register. The DMAx_X_MODIFY value is not applied when the inner loop count is ended by decrementing DMAx_CURR_X_COUNT from 1 to 0, except that it is applied on the final transfer when DMAx_CURR_Y_COUNT is 1 and DMAx_CURR_X_COUNT decrements from 1 to 0.

The DMAx_Y_MODIFY value is the byte-address increment that is applied after each decrement of the DMAx_CURR_Y_COUNT register. However, the

DMAx_Y_MODIFY value is not applied to the last item in the array on which the outer loop count (DMAx_CURR_Y_COUNT) also expires by decrementing from 1 to 0.

After the last transfer completes, DMAx_CURR_Y_COUNT = 1, DMAx_CURR_X_COUNT = 0, and DMAx_CURR_ADDR is equal to the last item's address plus DMAx_X_MODIFY.

If the DMA channel is programmed to refresh automatically (auto-buffer mode), then these registers will be loaded from DMAx_X_COUNT, DMAx_Y_COUNT, and DMAx_START_ADDR upon the first data transfer.

The DI_SEL configuration bit enables DMA interrupt requests every time the inner loop rolls over. If DI_SEL is cleared, but DI_EN is still set, only one interrupt is generated after the outer loop completes.

Examples of Two-Dimensional DMA

Example 1: Retrieve a 16 × 8 block of bytes from a video frame buffer of size (N × M) pixels:

DMAx_X_MODIFY = 1
DMAx_X_COUNT = 16
DMAx_Y_MODIFY = N-15 (offset from the end of one row to the start of another)
DMAx_Y_COUNT = 8

This produces the following address offsets from the start address:

```
0,1,2,...15,
N,N + 1, ... N + 15,
2N, 2N + 1,... 2N + 15, ...
7N, 7N + 1,... 7N + 15,
```

Example 2: Receive a video datastream of bytes,
(R,G,B pixels) × (N × M image size):

```
DMAx_X_MODIFY = (N * M)
DMAx_X_COUNT = 3
DMAx_Y_MODIFY = 1 - 2(N * M) (negative)
DMAx_Y_COUNT = (N * M)
```

This produces the following address offsets from the start address:

```
0, (N * M), 2(N * M),
1, (N * M) + 1, 2(N * M) + 1,
2, (N * M) + 2, 2(N * M) + 2,
    ...
(N * M) - 1, 2(N * M) - 1, 3(N * M) - 1,
```

Descriptor-based DMA Operation

In descriptor-based DMA operation, software does not set up DMA sequences by writing directly into DMA controller registers. Rather, software keeps DMA configurations, called descriptors, in memory. On demand, the DMA controller loads the descriptor from memory and overwrites the affected DMA registers by its own control. Descriptors can be fetched from L1 memory using the DCB bus or from external memory using the DEB bus.

A descriptor describes what kind of operation should be performed next by the DMA channel. This includes the DMA configuration word as well as data source/destination address, transfer count, and address modify values. A DMA sequence controlled by one descriptor is called a work unit.

Optionally, an interrupt can be requested at the end of any work unit by setting the `DI_EN` bit in the configuration word of the respective descriptor.

A DMA channel is started in descriptor-based mode by first writing the 32-bit address of the first descriptor into the DMAx_NEXT_DESC_PTR register (or the DMAx_CURR_DESC_PTR in case of descriptor array mode) and then performing a write to the DMAx_CONFIG register that sets the FLOW field to either 0x4, 0x6, or 0x7 and enables the DMAEN bit. This causes the DMA controller to immediately fetch the descriptor from the address pointed to by the DMAx_NEXT_DESC_PTR register. The fetch overwrites the DMAx_CONFIG register again. If the DMAEN bit is still set, the channel starts DMA processing.

The DFETCH bit in the DMAx_IRQ_STATUS register tells whether a descriptor fetch is ongoing on the respective DMA channel. The DMAx_CURR_DESC_PTR points to the descriptor value that is to be fetched next.

Descriptor List Mode

Descriptor list mode is selected by setting the FLOW bit field in the DMA channel's DMAx_CONFIG register to either 0x6 (small descriptor mode) or 0x7 (large descriptor mode). In either of these modes multiple descriptors form a chained list. Every descriptor contains a pointer to the next descriptor. When the descriptor is fetched, this pointer value is loaded into the DMAx_NEXT_DESC_PTR register of the DMA channel. In large descriptor mode this pointer is 32 bits wide. Therefore, the next descriptor may reside in any address space accessible through the DCB and DEB buses. In small descriptor mode this pointer is just 16 bits wide. For this reason, the next descriptor must reside in the same 64K byte address space as the first one because the upper 16 bits of the DMAx_NEXT_DESC_PTR register are not updated.

Descriptor list modes are started by writing first to the DMAx_NEXT_DESC_PTR register and then to the DMAx_CONFIG register.

Descriptor Array Mode

Descriptor array mode is selected by setting the FLOW bit field in the DMA channel's DMAx_CONFIG register to 0x4. In this mode, the descriptors do not contain further descriptor pointers. The initial DMAx_CURR_DESC_PTR value is written by software. It points to an array of descriptors. The individual descriptors are assumed to reside next to each other and, therefore, their addresses are known.

Variable Descriptor Size

In any descriptor-based mode the NDSIZE field in the configuration word specifies how many 16-bit words of the next descriptor need to be loaded on the next fetch. In descriptor-based operation, NDSIZE must be non-zero. The descriptor size can be any value from one entry (the lower 16 bits of DMAx_START_ADDR only) to nine entries (all the DMA parameters). Table 6-1 illustrates how a descriptor must be structured in memory. The values have the same order as the corresponding MMR addresses.

If, for example, a descriptor is fetched in array mode with NDSIZE = 0x5, the DMA controller fetches the 32-bit start address, the DMA configuration word, and the XCNT and XMOD values. However, it does not load YCNT and YMOD. This might be the case if the DMA operates in one-dimensional mode or if the DMA is in two-dimensional mode, but the YCNT and YMOD values do not need to change.

All the other registers not loaded from the descriptor retain their prior values, although the DMAx_CURR_ADDR, DMAx_CURR_X_COUNT, and DMAx_CURR_Y_COUNT registers are reloaded between the descriptor fetch and the start of DMA operation.

Table 6-1 shows the offsets for descriptor elements in the three modes described above. Note the names in the table describe the descriptor elements in memory, not the actual MMRs into which they are eventually

loaded. For more information regarding descriptor element acronyms, see Table 6-4 on page 6-65.

Table 6-1. Parameter Registers and Descriptor Offsets

Descriptor Offset	Descriptor Array Mode	Small Descriptor List Mode	Large Descriptor List Mode
0x0	SAL	NDPL	NDPL
0x2	SAH	SAL	NDPH
0x4	DMACFG	SAH	SAL
0x6	XCNT	DMACFG	SAH
0x8	XMOD	XCNT	DMACFG
0xA	YCNT	XMOD	XCNT
0xC	YMOD	YCNT	XMOD
0xE		YMOD	YCNT
0x10			YMOD

Note that every descriptor fetch consumes bandwidth from either the DCB bus or the DEB bus and the external memory interface, so it is best to keep the size of descriptors as small as possible.

Mixing Flow Modes

The FLOW mode of a DMA is not a global setting. If the DMA configuration word is reloaded with a descriptor fetch, the FLOW and NDSIZE bit fields can also be altered. A small descriptor might be used to loop back to the first descriptor if a descriptor array is used in an endless manner. If the descriptor chain is not endless and the DMA is required to stop after a certain descriptor has been processed, the last descriptor is typically processed in stop mode. That is, its FLOW and NDSIZE fields are 0, but its DMAEN bit is still set.

Functional Description

The following sections provide a functional description of DMA.

DMA Operation Flow

Figure 6-1 and Figure 6-2 describe the DMA flow.

DMA Startup

This section discusses starting DMA "from scratch." This is similar to starting it after it has been paused by the FLOW = 0 mode.

Before initiating DMA for the first time on a given channel, all parameter registers must be initialized. Be sure to initialize the upper 16 bits of the DMAx_NEXT_DESC_PTR (or DMAx_CURR_DESC_PTR register in FLOW = 4 mode) and DMAx_START_ADDR registers, because they might not otherwise be accessed, depending upon the flow mode. Also note that the DMAx_X_MODIFY and DMAx_Y_MODIFY registers are not preset to a default value at reset.

The user may wish to write other DMA registers that might be static during DMA activity (for example, DMAx_X_MODIFY, DMAx_Y_MODIFY). The contents of NDSIZE and FLOW in DMAx_CONFIG indicate which registers, if any, are fetched from descriptor elements in memory. After the descriptor

fetch, if any, is completed, DMA operation begins, initiated by writing DMAx_CONFIG with DMAEN = 1.

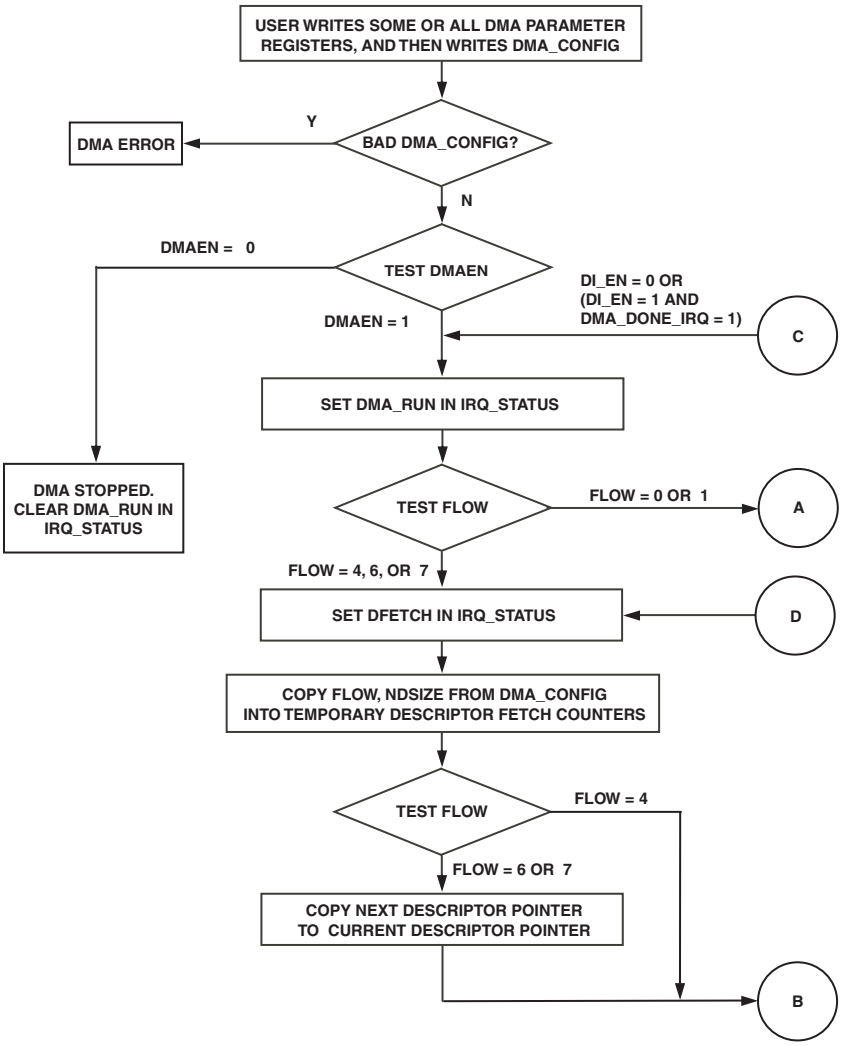

USER WRITES SOME OR ALL DMA PARAMETER REGISTERS, AND THEN WRITES DMA_CONFIG
BAD DMA_CONFIG?
Y
DMA ERROR
N
TEST DMAEN
DMAEN = 0
DMAEN = 1
DI_EN = 0 OR (DI_EN = 1 AND DMA_DONE_IRQ = 1)
C
SET DMA_RUN IN IRQ_STATUS
DMA STOPPED. CLEAR DMA_RUN IN IRQ_STATUS
TEST FLOW
FLOW = 0 OR 1
A
FLOW = 4, 6, OR 7
SET DFETCH IN IRQ_STATUS
D
COPY FLOW, NDSIZE FROM DMA_CONFIG INTO TEMPORARY DESCRIPTOR FETCH COUNTERS
TEST FLOW
FLOW = 4
FLOW = 6 OR 7
COPY NEXT DESCRIPTOR POINTER TO CURRENT DESCRIPTOR POINTER
B

Figure 6-1. DMA Flow, From DMA Controller's Point of View (1 of 2)

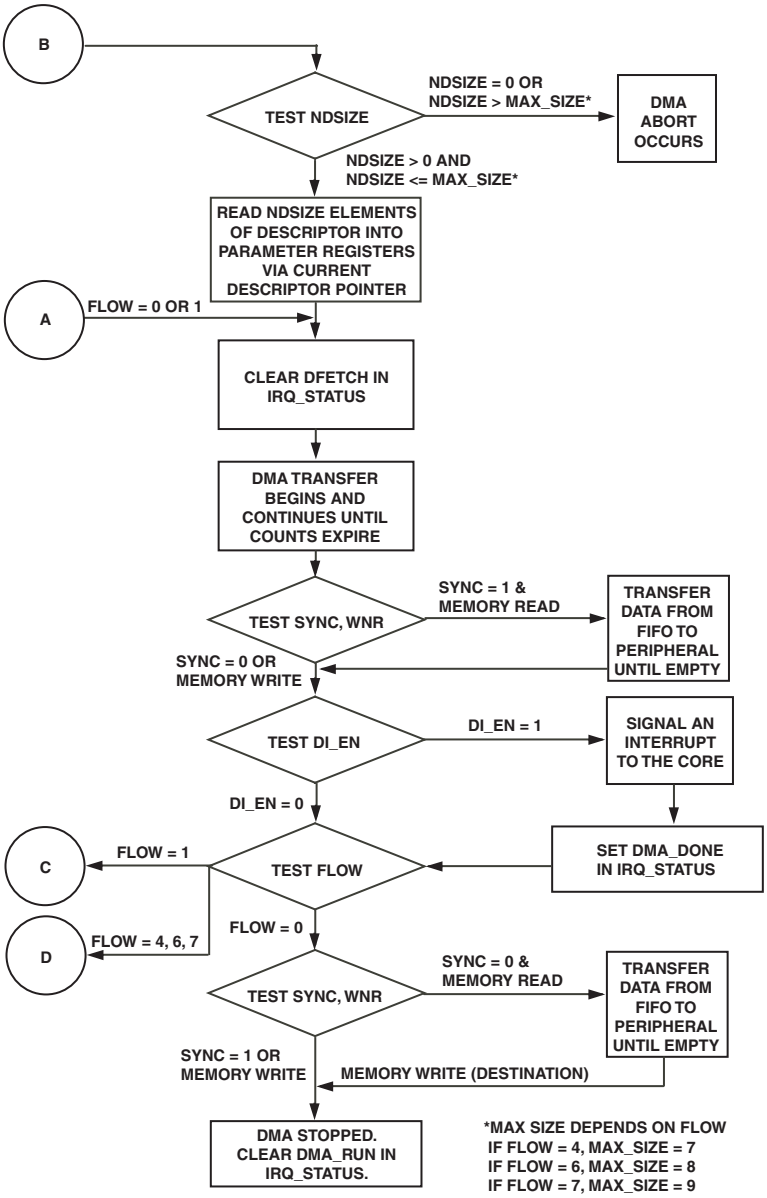


Figure 6-2. DMA Flow, From DMA Controller's Point of View (2 of 2)

When DMAx_CONFIG is written directly by software, the DMA controller recognizes this as the special startup condition that occurs when starting DMA for the first time on this channel or after the engine has been stopped (FLOW = 0).

When the descriptor fetch is complete and DMAEN = 1, the DMACFG descriptor element that was read into DMAx_CONFIG assumes control. Before this point, the direct write to DMAx_CONFIG had control. In other words, the WDSIZE, DI_EN, DI_SEL, SYNC, and DMA2D fields will be taken from the DMACFG value in the descriptor read from memory, while these field values initially written to the DMAx_CONFIG register are ignored.

As Figure 6-1 on page 6-19 and Figure 6-2 on page 6-20 show, at startup the FLOW and NDSIZE bits in DMAx_CONFIG determine the course of the DMA setup process. The FLOW value determines whether to load more current registers from descriptor elements in memory, while the NDSIZE bits detail how many descriptor elements to fetch before starting DMA. DMA registers not included in the descriptor are not modified from their prior values.

If the FLOW value specifies small or large descriptor list modes, the DMAx_NEXT_DESC_PTR is copied into DMAx_CURR_DESC_PTR. Then, fetches of new descriptor elements from memory are performed, indexed by DMAx_CURR_DESC_PTR, which is incremented after each fetch. If NDPL and/or NDPH is part of the descriptor, then these values are loaded into DMAx_NEXT_DESC_PTR, but the fetch of the current descriptor continues using DMAx_CURR_DESC_PTR. After completion of the descriptor fetch, DMAx_CURR_DESC_PTR points to the next 16-bit word in memory past the end of the descriptor.

If neither NDPH nor NDPL are part of the descriptor (that is, in descriptor array mode, FLOW = 4), then the transfer from NDPH/NDPL into DMAx_CURR_DESC_PTR does not occur. Instead, descriptor fetch indexing begins with the value in DMAx_CURR_DESC_PTR.

If DMACFG is not part of the descriptor, the previous DMAx_CONFIG settings (as written by MMR access at startup) control the work unit operation. If DMACFG is part of the descriptor, then the DMAx_CONFIG value programmed by the MMR access controls only the loading of the first descriptor from memory. The subsequent DMA work operation is controlled by the low byte of the descriptor's DMACFG and by the parameter registers loaded from the descriptor. The bits DI_EN, DI_SEL, DMA2D, WDSIZE, and WNR in the value programmed by the MMR access are disregarded.

The DMA_RUN and DFETCH status bits in the DMAx_IRQ_STATUS register indicate the state of the DMA channel. After a write to DMAx_CONFIG, the DMA_RUN and DFETCH bits can be automatically set to 1. No data interrupts are signaled as a result of loading the first descriptor from memory.

After the above steps, DMA data transfer operation begins. The DMA channel immediately attempts to fill its FIFO, subject to channel priority—a memory write (RX) DMA channel begins accepting data from its peripheral, and a memory read (TX) DMA channel begins memory reads, provided the channel wins the grant for bus access.

When the DMA channel performs its first data memory access, its address and count computations take their input operands from the start registers (DMAx_START_ADDR, DMAx_X_COUNT, DMAx_Y_COUNT), and write results back to the current registers (DMAx_CURR_ADDR, DMAx_CURR_X_COUNT, DMAx_CURR_Y_COUNT). Note also that the current registers are not valid until the first memory access is performed, which may be some time after the channel is started by the write to the DMA_CONFIG register. The current registers are loaded automatically from the appropriate descriptor elements, overwriting their previous contents, as follows.

- DMAx_START_ADDR is copied to DMAx_CURR_ADDR
- DMAx_X_COUNT is copied to DMAx_CURR_X_COUNT
- DMAx_Y_COUNT is copied to DMAx_CURR_Y_COUNT

Then DMA data transfer operation begins, as shown in Figure 6-2 on page 6-20.

DMA Refresh

On completion of a work unit:

- The DMA controller completes the transfer of all data between memory and the DMA unit.
- If SYNC = 1 and WNR = 0 (memory read), the DMA controller selects a synchronized transition and transfers all data to the peripheral before continuing.
- If enabled by DI_EN, the DMA controller signals an interrupt to the core and sets the DMA_DONE bit in the channel's DMAx_IRQ_STATUS register.
- If FLOW = 0 the DMA controller stops operation by clearing the DMA_RUN bit in DMAx_IRQ_STATUS register after all data in the channel's DMA FIFO has been transferred to the peripheral.
- During the fetch in FLOW modes 4, 6, and 7, the DMA controller sets the DFETCH bit in DMAx_IRQ_STATUS register to 1. At this point, the DMA operation depends on whether FLOW = 4, 6, or 7, as follows:

  If FLOW = 4 (descriptor array) the DMA controller loads a new descriptor from memory into the DMA registers using the contents of DMAx_CURR_DESC_PTR, and increments DMAx_CURR_DESC_PTR. The descriptor size comes from the NDSIZE field of the DMAx_CONFIG register prior to the beginning of the fetch.

  If FLOW = 6 (small descriptor list) the DMA controller copies the 32-bit DMAx_NEXT_DESC_PTR into DMAx_CURR_DESC_PTR. Next, the DMA controller fetches a descriptor from memory into the DMA

registers using the new contents of DMAx_CURR_DESC_PTR, and increments DMAx_CURR_DESC_PTR. The first descriptor element that is loaded is a new 16-bit value for the lower 16 bits of DMAx_NEXT_DESC_PTR, followed by the rest of the descriptor elements. The high 16 bits of DMAx_NEXT_DESC_PTR will retain their former value. This supports a shorter, more efficient descriptor than the large descriptor list model, which is suitable whenever the application can place the channel's descriptors in the same 64K byte range of memory.

If FLOW = 7 (large descriptor list) the DMA controller copies the 32-bit DMAx_NEXT_DESC_PTR into DMAx_CURR_DESC_PTR. Next, the DMA controller fetches a descriptor from memory into the DMA registers using the new contents of DMAx_CURR_DESC_PTR, and increments DMAx_CURR_DESC_PTR. The first descriptor element that is loaded is a new 32-bit value for the full DMAx_NEXT_DESC_PTR, followed by the rest of the descriptor elements. The high 16 bits of DMAx_NEXT_DESC_PTR may differ from their former value. This supports a fully flexible descriptor list which can be located anywhere in internal memory or external memory.

- If it is necessary to link from a descriptor chain whose descriptors are in one 64K byte area to another chain whose descriptors are outside that area, only the descriptor containing the link to the new 64K byte range needs to use FLOW = 7. All descriptors that reference the same 64K byte area may use FLOW = 6.
- If FLOW = 4, 6, or 7 (descriptor array, small descriptor list, or large descriptor list, respectively), the DMA controller clears the DFETCH bit in the DMAx_IRQ_STATUS register.
- If FLOW = any value but 0 (Stop), the DMA controller begins the next work unit for that channel, which must contend with other channels for priority on the memory buses. On the first memory transfer of the new work unit, the DMA controller updates the cur-

rent registers from the start registers:

DMAx_CURR_ADDR loaded from DMAx_START_ADDR
DMAx_CURR_X_COUNT loaded from DMAx_X_COUNT
DMAx_CURR_Y_COUNT loaded from DMAx_Y_COUNT

The DFETCH bit in the DMAx_IRQ_STATUS register is then cleared, after which the DMA transfer begins again, as shown in Figure 6-2 on page 6-20.

Work Unit Transitions

Transitions from one work unit to the next are controlled by the SYNC bit in the DMAx_CONFIG register of the work units. In general, continuous transitions have lower latency at the cost of restrictions on changes of data format or addressed memory space in the two work units. These latency gains and data restrictions arise from the way the DMA FIFO pipeline is handled while the next descriptor is fetched. In continuous transitions (SYNC = 0), the DMA FIFO pipeline continues to transfer data to and from the peripheral or destination memory during the descriptor fetch and/or when the DMA channel is paused between descriptor chains.

Synchronized transitions (SYNC = 1), on the other hand, provide better real-time synchronization of interrupts with peripheral state and greater flexibility in the data formats and memory spaces of the two work units, at the cost of higher latency in the transition. In synchronized transitions, the DMA FIFO pipeline is drained to the destination or flushed (RX data discarded) between work units.

Work unit transitions for MDMA streams are controlled by the SYNC bit of the MDMA source channel's DMAx_CONFIG register. The SYNC bit of the MDMA destination channel is reserved and must be 0. In transmit (memory read) channels, the SYNC bit of the last descriptor prior to the transition controls the transition behavior. In contrast, in receive channels, the SYNC bit of the first descriptor of the next descriptor chain controls the transition.

DMA Transmit and MDMA Source

In DMA transmit (memory read) and MDMA source channels, the SYNC bit controls the interrupt timing at the end of the work unit and the handling of the DMA FIFO between the current and next work units.

If SYNC = 0, a continuous transition is selected. In a continuous transition, just after the last data item is read from memory, the following operations start in parallel:

- The interrupt (if any) is signalled.
- The DMA_DONE bit in the DMAx_IRQ_STATUS register is set.
- The next descriptor begins to be fetched.
- The final data items are delivered from the DMA FIFO to the destination memory or peripheral.

This allows the DMA to provide data from the FIFO to the peripheral "continuously" during the descriptor fetch latency period.

When SYNC = 0, the final interrupt (if enabled) occurs when the last data is read from memory. This interrupt is at the earliest time that the output memory buffer may safely be modified without affecting the previous data transmission. Up to four data items may still be in the DMA FIFO, however, and not yet at the peripheral, so the DMA interrupt should not be used as the sole means of synchronizing the shutdown or reconfiguration of the peripheral following a transmission.

If SYNC = 0 (continuous transition) on a transmit (memory read) descriptor, the next descriptor must have the same data word size, read/write direction, and source memory (internal vs. external) as the current descriptor.

SYNC = 0 selects continuous transition on a work unit in FLOW = 0 mode with interrupt enabled. The interrupt service routine may begin execution while the final data is still draining from the FIFO to the peripheral. This

is indicated by the DMA_RUN bit in the DMAx_IRQ_STATUS register; if it is 1, the FIFO is not empty yet. Do not start a new work unit with different word size or direction while DMA_RUN = 1. Further, if the channel is disabled (by writing DMAEN = 0), the data in the FIFO is lost.

SYNC = 1 selects a synchronized transition in which the DMA FIFO is first drained to the destination memory or peripheral before any interrupt is signalled and before any subsequent descriptor or data is fetched. This incurs greater latency, but provides direct synchronization between the DMA interrupt and the state of the data at the peripheral.

For example, if SYNC = 1 and DI_EN = 1 on the last descriptor in a work unit, the interrupt occurs when the final data has been transferred to the peripheral, allowing the service routine to properly switch to non-DMA transmit operation. When the interrupt service routine is invoked, the DMA_DONE bit is set and the DMA_RUN bit is cleared.

A synchronized transition also allows greater flexibility in the format of the DMA descriptor chain. If SYNC = 1, the next descriptor may have any word size or read/write direction supported by the peripheral and may come from either memory space (internal or external). This can be useful in managing MDMA work unit queues, since it is no longer necessary to interrupt the queue between dissimilar work units.

DMA Receive

In DMA receive (memory write) channels, the SYNC bit controls the handling of the DMA FIFO between descriptor chains (not individual descriptors), when the DMA channel is paused. The DMA channel pauses after descriptors with FLOW = 0 mode, and may be restarted (for example, after an interrupt) by writing the channel's DMAx_CONFIG register with DMAEN = 1.

If the SYNC bit is 0 in the new work unit's DMAx_CONFIG value, a continuous transition is selected. In this mode, any data items received into the DMA FIFO while the channel was paused are retained, and they are the first

items written to memory in the new work unit. This mode of operation provides lower latency at work unit transitions and ensures that no data items are dropped during a DMA pause, at the cost of certain restrictions on the DMA descriptors.

If the SYNC bit is 0 on the first descriptor of a descriptor chain after a DMA pause, the DMA word size of the new chain must not change from the word size of the previous descriptor chain active before the pause, unless the DMA channel is reset between chains by writing the DMAEN bit to 0 and then to 1 again.

If the SYNC bit is 1 in the new work unit's DMAx_CONFIG value, a synchronized transition is selected. In this mode, only the data received from the peripheral by the DMA channel after the write to the DMAx_CONFIG register are delivered to memory. Any prior data items transferred from the peripheral to the DMA FIFO before this register write are discarded. This provides direct synchronization between the data stream received from the peripheral and the timing of the channel restart (when the DMAx_CONFIG register is written).

For receive DMAs, the SYNC bit has no effect in transitions between work units in the same descriptor chain (that is, when the previous descriptor's FLOW mode was not 0, so that DMA channel did not pause.)

If a descriptor chain begins with a SYNC bit of 1, there is no restriction on DMA word size of the new chain in comparison to the previous chain.

The DMA word size must not change between one descriptor and the next in any DMA receive (memory write) channel within a single descriptor chain, regardless of the SYNC bit setting. In other words, if a descriptor has WNR = 1 and FLOW = 4, 6, or 7, then the next descriptor must have the same word size. For any DMA receive (memory write) channel, there is no restriction on changes of memory space (internal vs. external) between descriptors or

descriptor chains. DMA transmit (memory read) channels may have such restrictions (see "DMA Transmit and MDMA Source" on page 6-26).

Stopping DMA Transfers

In FLOW = 0 mode, DMA stops automatically after the work unit is complete.

If a list or array of descriptors is used to control DMA, and if every descriptor contains a DMACFG element, then the final DMACFG element should have a FLOW = 0 setting to gracefully stop the channel.

In autobuffer (FLOW = 1) mode, or if a list or array of descriptors without DMACFG elements is used, then the DMA transfer process must be terminated by an MMR write to the DMAx_CONFIG register with a value whose DMAEN bit is 0. A write of 0 to the entire register will always terminate DMA gracefully (without DMA abort).

If a channel has been stopped abruptly by writing DMAx_CONFIG to 0 (or any value with DMAEN = 0), the user must ensure that any memory read or write accesses in the pipelines have completed before enabling the channel again. If the channel is enabled again before an "orphan" access from a previous work unit completes, the state of the DMA interrupt and FIFO is unspecified. This can generally be handled by ensuring that the core allocates several consecutive idle cycles in its usage of the relevant memory space to allow up to three pending DMA accesses to issue, plus allowing enough memory access time for the accesses themselves to complete.

DMA Errors (Aborts)

The DMA controller flags conditions that cause the DMA process to end abnormally (abort). This functionality is provided as a tool for system development and debug to detect DMA-related programming errors. DMA errors (aborts) are detected by the DMA channel module in the

cases listed below. When a DMA error occurs, the channel is immediately stopped (DMA_RUN goes to 0) and any prefetched data is discarded. In addition, a DMA_ERROR interrupt is asserted.

There is only one DMA_ERROR interrupt for the whole DMA controller, which is asserted whenever any of the channels has detected an error condition.

The DMA_ERROR interrupt handler must:

- Read each channel's DMAx_IRQ_STATUS register to look for a channel with the DMA_ERR bit set (bit 1).
- Clear the problem with that channel (for example, fix register values).
- Clear the DMA_ERR bit (write DMAx_IRQ_STATUS with bit 1 set).

The following error conditions are detected by the DMA hardware and result in a DMA abort interrupt.

- The configuration register contains invalid values:
  - Incorrect WDSIZE value (WDSIZE = b#11)
  - Bit 15 not set to 0
  - Incorrect FLOW value (FLOW = 2, 3, or 5)
  - NDSIZE value does not agree with FLOW. See Table 6-2 on page 6-32.
- A disallowed register write occurred while the channel was running. Only the DMAx_CONFIG and DMAx_IRQ_STATUS registers can be written when DMA_RUN = 1.

- An address alignment error occurred during any memory access. For example, when DMAx_CONFIG register WDSIZE = 1 (16-bit) but the least significant bit (LSB) of the address is not equal to b#0, or when WDSIZE = 2 (32-bit) but the two LSBs of the address are not equal to b#00.
- A memory space transition was attempted (internal-to-external or vice versa). For example, the value in the DMAx_CURR_ADDR register or DMAx_CURR_DESC_PTR register crossed a memory boundary.
- A memory access error occurred. Either an access attempt was made to an internal address not populated or defined as cache, or an external access caused an error (signaled by the external memory interface).

Some prohibited situations are not detected by the DMA hardware. No DMA abort is signaled for these situations:

- DMAx_CONFIG direction bit (WNR) does not agree with the direction of the mapped peripheral.
- DMAx_CONFIG direction bit does not agree with the direction of the MDMA channel.
- DMAx_CONFIG word size (WDSIZE) is not supported by the mapped peripheral. See Table 6-2 on page 6-32.
- DMAx_CONFIG word size in source and destination of the MDMA stream are not equal.

- Descriptor chain indicates data buffers that are not in the same internal/external memory space.
- In 2-D DMA, `X_COUNT` = 1

Table 6-2. Legal NDSIZE Values

FLOW	NDSIZE	Note
0	0	
1	0	
4	0 < NDSIZE ≤ 7	Descriptor array, no descriptor pointer fetched
6	0 < NDSIZE ≤ 8	Descriptor list, small descriptor pointer fetched
7	0 < NDSIZE ≤ 9	Descriptor list, large descriptor pointer fetched

DMA Control Commands

Advanced peripherals, such as an Ethernet MAC module, are capable of managing some of their own DMA operations, thus dramatically improving real-time performance and relieving control and interrupt demands on the Blackfin processor core. These peripherals may communicate to the DMA controller using DMA control commands, which are transmitted from the peripheral to the associated DMA channel over internal DMA request buses. Refer to "Unique Behavior for the ADSP-BF52x Processor" on page 6-105 to determine if DMA control commands are applicable to a particular product.

The request buses consist of three wires per DMA-management-capable peripheral. The DMA control commands extend the set of operations available to the peripheral beyond the simple "request data" command used by peripherals in general.

While these DMA control commands are not visible to or controllable by the user, their use by a peripheral has implications for the structure of the DMA transfers which that peripheral can support. It is important that

application software be written to comply with certain restrictions regarding work units and descriptor chains (described later in this section) so that the peripheral operates properly whenever it issues DMA control commands.

MDMA channels do not service peripherals and therefore do not support DMA control commands. The DMA control commands are shown in Table 6-3.

Table 6-3. DMA Control Commands

Code	Name	Description
000	NOP	No operation
001	Restart	Restarts the current work unit from the beginning
010	Finish	Finishes the current work unit and starts the next
011	-	Reserved
100	Req Data	Typical DMA data request
101	Req Data Urgent	Urgent DMA data request
110	-	Reserved
111	-	Reserved

Additional information for the control commands includes:

- **Restart**

  The `Restart` command causes the current work unit to interrupt processing and start over, using the addresses and counts from `DMAx_START_ADDR`, `DMAx_X_COUNT`, and `DMAx_Y_COUNT`. No interrupt is signalled.

  If a channel programmed for transmit (memory read) receives a `Restart` command, the channel momentarily pauses while any pending memory reads initiated prior to the `Restart` command are completed.

During this period of time, the channel does not grant DMA requests. Once all pending reads have been flushed from the channel's pipelines, the channel resets its counters and FIFO and starts prefetch reads from memory. DMA data requests from the peripheral are granted as soon as new prefetched data is available in the DMA FIFO. The peripheral can thus use the `Restart` command to re-attempt a failed transmission of a work unit.

If a channel programmed for receive (memory write) receives a `Restart` command, the channel stops writing to memory, discards any data held in its DMA FIFO, and resets its counters and FIFO. As soon as this initialization is complete, the channel again grants DMA write requests from the peripheral. The peripheral can thus use the `Restart` command to abort transfer of received data into a work unit and re-use the memory buffer for a later data transfer.

- **Finish**

  The `Finish` command causes the current work unit to terminate and move on to the next work unit. An interrupt is signalled as usual, if selected by the `DI_EN` bit. The peripheral can thus use the `Finish` command to partition the DMA stream into work units on its own, perhaps as a result of parsing the data currently passing though its supported communication channel, without direct real-time control by the processor.

  If a channel programmed for transmit (memory read) receives a `Finish` command, the channel momentarily pauses while any pending memory reads initiated prior to the `Finish` command are completed. During this period of time, the channel does not grant DMA requests. Once all pending reads have been flushed from the channel's pipelines, the channel signals an interrupt (if enabled), and begins fetching the next descriptor (if any). DMA data requests from the peripheral are granted as soon as new prefetched data is available in the DMA FIFO.

If a channel programmed for receive (memory write) receives a `Finish` command, the channel stops granting new DMA requests while it drains its FIFO. Any DMA data received by the DMA controller prior to the `Finish` command is written to memory. When the FIFO reaches an empty state, the channel signals an interrupt (if enabled) and begins fetching the next descriptor (if any). Once the next descriptor has been fetched, the channel initializes its FIFO and then resumes granting DMA requests from the peripheral.

- **Request Data**

  The `Request Data` command is identical to the DMA request operation of peripherals that are not DMA-management-capable.

- **Request Data Urgent**

  The `Request Data Urgent` command behaves identically to the DMA `Request` command, except that the DMA channel performs its memory accesses with urgent priority while it is asserted. This includes both data and descriptor-fetch memory accesses. A DMA-management-capable peripheral might use this command if an internal FIFO is approaching a critical condition.

Restrictions

The proper operation of the 4-location DMA channel FIFO leads to certain restrictions in the sequence of DMA control commands.

Transmit Restart or Finish

No `Restart` or `Finish` command may be issued by a peripheral to a channel configured for memory read unless the peripheral has already performed at least one DMA transfer in the current work unit and the current work unit has more than four items remaining in `DMAx_CURR_X_COUNT`/ `DMAx_CURR_Y_COUNT` (thus not yet read from mem-

ory). Otherwise, the current work unit may already have completed memory operations and can no longer be restarted or finished properly.

If the DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT value of the current work unit is sufficiently large that it is always at least five more than the maximum data count prior to any Restart or Finish command, the above restriction is satisfied. This implies that any work unit which might be managed by Restart or Finish commands must have DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT values representing at least five data items.

Particularly if the DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT registers are programmed to 0 (representing 65,536 transfers, the maximum value) the channel will operate properly for 1-D work units up to 65,531 data items or 2-D work units up to 4,294,967,291 data items.

Receive Restart or Finish

No Restart or Finish command may be issued by a peripheral to a channel configured for memory write unless either the peripheral has already performed at least five DMA transfers in the current work unit or the previous work unit was terminated by a Finish command and the peripheral has performed at least one DMA transfer in the current work unit. If five data transfers have been performed, then at least one data item has been written to memory in the current work unit, which implies that the current work unit's descriptor fetch completed before the data grant of the fifth item. Otherwise, if less than five data items have been transferred, it is possible that all of them are still in the DMA FIFO and the previous work unit is still in the process of completion and transition between work units.

Similarly, if a Finish command ended the previous work unit and at least one subsequent DMA data transfer has occurred, then the fact that the DMA channel issued the grant guarantees that the previous work unit has already completed the process of draining its data to memory and transitioning to the new work unit.

If a peripheral terminates all work units with the Finish opcode (effectively assuming responsibility for all work unit boundaries for the DMA channel), then the peripheral need only ensure that it performs a single transfer in each work unit before any restart or finish. This requires, however, that the user programs the descriptors for all work units managed by the channel with DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT values representing more data items than the maximum work unit size that the peripheral will encounter. For example, DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT values of 0 allow the channel to operate properly on 1-D work units up to 65,535 data items and 2-D work units up to 4,294,967,295 data items.

Handshaked Memory DMA Operation

Handshaked memory DMA operation is not available for all products. Refer to the "Unique Behavior for the ADSP-BF52x Processor" on page 6-105 to determine whether this feature applies to this product.

Each DMARx input has its own set of control and status registers. Handshake operation for MDMA0 is enabled by the HMDMAEN bit in the HMDMA0_CONTROL register. Similarly, the HMDMAEN bit in the HMDMA1_CONTROL register enables handshake mode for MDMA1.

It is important to understand that the handshake hardware works completely independently from the descriptor and autobuffer capabilities of the MDMA, allowing most flexible combinations of logical data organization vs. data portioning as required by FIFO depths, for example. If, however, the connected device requires certain behavior of the address lines, these must be controlled by traditional DMA setup.

The HMDMA unit controls only the destination (memory write) channel of the memory DMA. The source channel (memory-read side) fills the 8-deep DMA buffers immediately after the receive side is enabled and issues eight read commands.

The HMDMAx_BCINIT registers control how many data transfers are performed upon every DMA request. If set to one, the peripheral can time every individual data transfer. If greater than one, the peripheral must have sufficient buffer size to provide or consume the number of words programmed. Once the transfer has been requested, no further handshake can hold off the DMA from transferring the entire block, except by stalling the EBIU accesses by the ARDY signal. Nevertheless, the peripheral may request a block transfer before the entire buffer is available by simply taking the minimum transfer time based on wait-state settings into consideration.

The block count defines how many data transfers are performed by the MDMA engine. A single DMA transfer can cause two read or write operations on the EBIU port if the transfer word size is set to 32-bit in the MDMA_yy_CONFIG register (WDSIZE = b#10).

Since the block count registers are 16 bits wide, blocks can group up to 65,535 transfers.

Once a block transfer has been started, the HMDMAx_BCOUNT registers return the remaining number of transfers to complete the current block. When the complete block has been processed, the HMDMAx_BCOUNT register returns zero. Software can force a reload of the HMDMAx_BCOUNT from the HMDMAx_BCINIT register even during normal operation by setting the RBC bit in the HMDMAx_CONTROL register. Set RBC when the HMDMA module is already active, but only when the MDMA is not enabled.

Pipelining DMA Requests

The device mastering the DMA request lines is allowed to request additional transfers even before the former transfer has completed. As long as the device can provide or consume sufficient data it is permitted to pulse the DMARx inputs multiple times.

The HMDMAx_ECOUNT registers are incremented every time a significant edge is detected on the respective DMARx input, and they are decremented when

the MDMA completes the block transfer. These read-only registers use a 16-bit twos-complement data representation: if they return zero, all requested block transfers have been performed. A positive value signals up to 32767 requests that haven't been served yet and indicates that the MDMA is currently processing. Negative values indicate the number of DMA requests that will be ignored by the engine. This feature restrains initial pulses on the DMARx inputs at startup.

The HMDMAx_ECINIT registers reload the HMDMAx_ECOUNT registers every time the handshake mode is enabled (when the HMDMAEN bit changes from 0 to 1). If the initial edge count value is 0, the handshake operation starts with a settled request budget. If positive, the engine starts immediately transferring the programmed number (up to 32767) of blocks once enabled, even without detecting any activity on the DMARx pins. If negative, the engine will disregard the programmed number (up to 32768) significant edges on the DMARx inputs before starting normal operation.

Figure 6-3 illustrates how an asynchronous FIFO could be connected. In such a scenario the REP bit should be cleared to let the DMARx request pin listen to falling edges. The Blackfin processor does not evaluate the full flag such FIFOs usually provide because asynchronous polling of that signal would reduce the system throughput drastically. Moreover, the processor first fills the FIFO by initializing the HMDMAx_ECINIT register to 1024, which equals the depth of the FIFO. Once enabled, the MDMA automatically transmits 1024 data words. Afterward it continues to trans-

mit only if the FIFO is emptied by its read strobe again. Most likely, the HMDMAx_BCINIT register is programmed to 1 in this case.

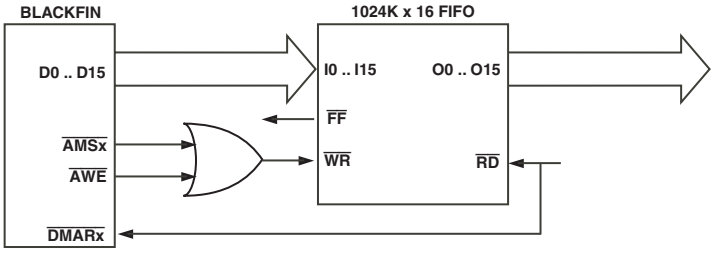


Figure 6-3. Transmit DMA Example Connection

In the receive example shown in Figure 6-4, the Blackfin processor again does not use the FIFO's internal control mechanism. Rather than testing the empty flag, the processor counts the number of data words available in the FIFO in its own HMDMAx_ECOUNT register. Theoretically, the MDMA could immediately process data as soon as it is written into the FIFO by the write strobe, but the fast MDMA engine would read out the FIFO quickly and stall soon if the FIFO was not promptly filled with new data. Streaming applications can balance the FIFO so that the producer is never held off by a full FIFO and the consumer is never held by an empty FIFO. This is accomplished by filling the FIFO halfway and then letting both consumer and producer run at the same speed. In this case the HMDMAx_ECINIT register can be written with a negative value, which corre-

sponds to half the FIFO depth. Then, the MDMA does not start consuming data as long as the FIFO is not half-filled.

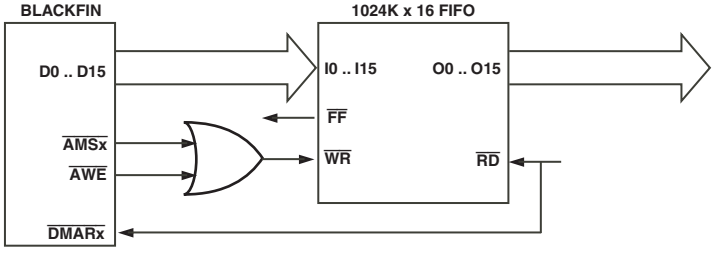


Figure 6-4. Receive DMA Example Connection

On internal system buses, memory DMA channels have lower priority than other DMAs. In busy systems, the memory DMAs may tend to starve. As this is not acceptable when transferring data through high-speed FIFOs, the handshake mode provides a high-water functionality to increase the MDMA's priority. With the UTE bit in the HMDMAx_CONTROL register set, the MDMA gets higher priority as soon as a (positive) value in the HMDMAx_ECOUNT register becomes higher than the threshold held by the HMDMAx_ECURGENT register.

HMDMA Interrupts

In addition to the normal MDMA interrupt channels, the handshake hardware provides two new interrupt sources for each DMARx input. The HMDMAx_CONTROL registers provide interrupt enable and status bits. The interrupt status bits require a write-1-to-clear operation to cancel the interrupt request.

The block done interrupt signals that a complete MDMA block, as defined by the HMDMAx_BCINIT register, has been transferred (when the HMDMAx_BCOUNT register decrements to zero). While the BDIE bit enables this interrupt, the MBDI bit can gate it until the edge count also becomes zero, meaning that all requested MDMA transfers have been completed.

The overflow interrupt is generated when the HMDMA_ECOUNT register overflows. Since it can count up to 32767, which is much more than most peripheral devices can support, the Blackfin processor has another threshold register called HMDMA_ECOVERFLOW. It resets to 0xFFFF and should be written with any positive value by the user before enabling the function by the OIE bit. Then, the overflow interrupt is issued when the value of the HMDMA_ECOUNT register exceeds the threshold in the HMDMA_ECOVERFLOW register.

DMA Performance

The DMA system is designed to provide maximum throughput per channel and maximum utilization of the internal buses, while accommodating the inherent latencies of memory accesses.

The Blackfin architecture features several mechanisms to customize system behavior for best performance. This includes DMA channel prioritization, traffic control, and priority treatment of bursted transfers. Nevertheless, the resulting performance of a DMA transfer often depends on application-level circumstances. For best performance consider the following system software architecture questions.

- What is the required DMA bandwidth?
- Which DMA transfers have real-time requirements and which do not?
- How heavily is the DMA controller competing with the core for on-chip and off-chip resources?
- How often do competing DMA channels require the bus systems to alter direction?
- How often do competing DMA or core accesses cause the SDRAM to open different pages?
- Is there a way to distribute DMA requests nicely over time?

A key feature of the DMA architecture is the separation of the activity on the DMA access bus (DAB) used by the peripherals from the activity on the buses between the DMA and memory. For DMA to/from on-chip memory the DMA core bus (DCB) is used, and the DMA external bus (DEB) is used for DMA transfers with off-chip memory. The “Chip Bus Hierarchy” chapter explains the bus architecture.

Each peripheral DMA channel has its own data FIFO which lies between the DAB bus and the memory buses. These FIFOs automatically prefetch data from memory for transmission and buffer received data for later memory writes. This allows the peripheral to be granted a DMA transfer with very low latency compared to the total latency of a pipelined memory access, permitting the repeat rate (bandwidth) of each DMA channel to be as fast as possible.

DMA Throughput

Each peripheral DMA channel has a maximum transfer rate of one 16-bit word per two system clocks in either direction. Like the DAB and DEB buses, the DMA controller resides in the SCLK domain. The controller synchronizes accesses to and from the DCB bus, which runs at the CCLK rate.

Each memory DMA channel has a maximum transfer rate of one 16-bit word per system clock (SCLK) cycle.

When the traffic on all DMA channels is taken in the aggregate:

- Transfers between the peripherals and the DMA unit have a maximum rate of one 16-bit transfer per system clock.
- Transfers between the DMA unit and internal memory (L1) have a maximum rate of one 16-bit transfer per system clock.
- Transfers between the DMA unit and external memory have a maximum rate of one 16-bit transfer per system clock.

Some considerations which limit the actual performance include:

- Accesses to internal or external memory which conflict with core accesses to the same memory. This can cause delays, for example when both the core and the DMA access the same L1 bank, when SDRAM pages need to be opened/closed, or when cache lines are filled.
- Direction changes from RX to TX on the DAB bus impose a one SCLK cycle delay.
- Direction changes on the DCB bus (for example, write followed by read) to the same bank of internal memory can impose delays.
- Direction changes (for example, read followed by write) on the DEB bus to external memory can each impose a several-cycle delay.
- MMR accesses to DMA registers other than DMAx_CONFIG, DMAx_IRQ_STATUS, or DMAx_PERIPHERAL_MAP stall all DMA activity for one cycle per 16-bit word transferred. In contrast, MMR accesses to the control/status registers do not cause stalls or wait states.
- Reads from DMA registers other than control/status registers use one PAB bus wait state, delaying the core for several core clocks.
- Descriptor fetches consume one DMA memory cycle per 16-bit word read from memory, but do not delay transfers on the DAB bus.
- Initialization of a DMA channel stalls DMA activity for one cycle. This occurs when DMAEN changes from 0 to 1 or when the SYNC bit is set in the DMAx_CONFIG register.

Several of these factors may be minimized by proper design of the application software. It is often possible to structure the software to avoid internal and external memory conflicts by careful allocation of data buffers within banks and pages, and by planning for low cache activity during

critical DMA operations. Furthermore, unnecessary MMR accesses can be minimized, especially by using descriptors or autobuffering.

Efficiency loss caused by excessive direction changes (thrashing) can be minimized by the processor's traffic control features, described in the next section.

The MDMA channels are clocked by SCLK. If the source and destination are in different memory spaces (one internal and one external), the internal and external memory transfers are typically simultaneous and continuous, maintaining 100% bus utilization of the internal and external memory interfaces. This performance is affected by core-to-system clock frequency ratios. At ratios below about 2.5:1, synchronization and pipeline latencies result in lower bus utilization in the system clock domain. For example DMA typically runs at 2/3 of the system clock rate when the core-to-system clock ratio is 2:1. At higher clock ratios, full bandwidth is maintained.

If the source and destination are in the same memory space (both internal or both external), the MDMA stream typically prefetches a burst of source data into the FIFO, and then automatically turns around and delivers all available data from the FIFO to the destination buffer. The burst length is dependent on traffic, and is equal to three plus the memory latency at the DMA in SCLKs (which is typically seven for internal transfers and six for external transfers).

Memory DMA Timing Details

When the destination DMAx_CONFIG register is written, MDMA operation starts after a latency of three SCLK cycles.

If either MDMA channel has been selected to use descriptors, the descriptors are fetched from memory. The destination channel descriptors are fetched first. Then the source MDMA channel begins fetching data from the source buffer, after a latency of four SCLK cycles after the last descriptor word is returned from memory. Due to memory pipelining, this is

typically eight SCLK cycles after the fetch of the last descriptor word. The resulting data is deposited in the MDMA channel's 8-location FIFO. After a latency of two SCLK cycles, the destination MDMA channel begins writing data to the destination memory buffer.

Static Channel Prioritization

DMA channels are ordinarily granted service strictly according to their priority. The priority of a channel is simply its channel number, where lower priority numbers are granted first. Thus, peripherals with high data rates or low latency requirements should be assigned to lower numbered (higher priority) channels using the PMAP field in the DMAx_PERIPHERAL_MAP registers. The memory DMA streams are always lower static priority than the peripherals, but as they request service continuously, they ensure that any time slots unused by peripheral DMA are applied to MDMA transfers.

Temporary DMA Urgency

Typically, DMA transfers for a given peripheral occur at regular intervals. Generally, the shorter the interval, the higher the priority that should be assigned to the peripheral. If the average bandwidth of all the peripherals is not too large a fraction of the total, then all peripherals' requests should be granted as required.

Occasionally, instantaneous DMA traffic might exceed the available bandwidth, causing congestion. This may occur if L1 or external memory is temporarily stalled, perhaps for an SDRAM page swap or a cache line fill. Congestion might also occur if one or more DMA channels initiates a flurry of requests, perhaps for descriptor fetches or to fill a FIFO in the DMA or in the peripheral.

If congestion persists, lower priority DMA peripherals may become starved for data. Even though the peripheral's priority is low, if the necessary data transfer does not take place before the end of the peripheral's regular interval, system failure may result. To minimize this possibility,

the DMA unit detects peripherals whose need for data has become urgent, and preferentially grants them service at the highest priority.

A DMA channel's request for memory service is defined as urgent if both:

- The channel's FIFO is not ready for a DAB bus transfer (that is, a transmit FIFO is empty or a receive FIFO is full), and
- The peripheral is asserting its DMA request line.

Descriptor fetches may be urgent if they are necessary to initiate or continue a DMA work unit chain for a starving peripheral.

DMA requests from an MDMA channel become urgent when handshaked operation is enabled and the DMARx edge count exceeds the value stored in the HMDMAx_ECURGENT register. If handshaked operation is disabled, software can control urgency of requests directly by altering the DRQ bit field in the HMDMAx_CONTROL register.

When one or more DMA channels express an urgent memory request, two events occur:

- All non-urgent memory requests are decreased in priority by 32, guaranteeing that only an urgent request will be granted. The urgent requests compete with each other, if there is more than one, and directional preference among urgent requests is observed.
- The resulting memory transfer is marked for expedited processing in the targeted memory system (L1 or external). All prior incomplete memory transfers ahead of it in that memory system are also marked for expedited processing. This may cause a series of external memory core accesses to be delayed for a few cycles so that a peripheral's urgent request may be accommodated.

The preferential handling of urgent DMA transfers is completely automatic. No user controls are required for this function to operate.

Memory DMA Priority and Scheduling

All MDMA operations have lower precedence than any peripheral DMA operations. MDMA thus makes effective use of any memory bandwidth unused by peripheral DMA traffic.

By default, when more than one MDMA stream is enabled and ready, only the highest priority MDMA stream is granted. If it is desirable for the MDMA streams to share the available bandwidth, the MDMA_ROUND_ROBIN_PERIOD may be programmed to select each stream in turn for a fixed number of transfers.

If two MDMA streams are used (S0-D0 and S1-D1), the user may choose to allocate bandwidth either by fixed stream priority or by a round-robin scheme. This is selected by programming the MDMA_ROUND_ROBIN_PERIOD field in the DMA_TC_PER register (see "Static Channel Prioritization" on page 6-46).

If this field is set to 0, then MDMA is scheduled by fixed priority. MDMA stream 0 takes precedence over MDMA stream 1 whenever stream 0 is ready to perform transfers. Since an MDMA stream is typically capable of transferring data on every available cycle, this could cause MDMA stream 1 traffic to be delayed for an indefinite time until any and all MDMA stream 0 operations are completed. This scheme could be appropriate in systems where low duration but latency-sensitive data buffers need to be moved immediately, interrupting long duration, low priority background transfers.

If the MDMA_ROUND_ROBIN_PERIOD field is set to some nonzero value in the range $1 \leq P \leq 31$, then a round-robin scheduling method is used. The two MDMA streams are granted bus access in alternation in bursts of up to P data transfers. This could be used in systems where two transfer processes need to coexist, each with a guaranteed fraction of the available bandwidth. For example, one stream might be programmed for internal-to-external moves while the other is programmed for exter-

nal-to-internal moves, and each would be allocated approximately equal data bandwidth.

In round-robin operation, the MDMA stream selection at any time is either “free” or “locked.” Initially, the selection is free. On any free cycle available to MDMA (when no peripheral DMA accesses take precedence), if either or both MDMA streams request access, the higher precedence stream will be granted (stream 0 in case of conflict), and that stream’s selection is then “locked.” The MDMA_ROUND_ROBIN_COUNT counter field in the DMA_TC_CNT register is loaded with the period P from MDMA_ROUND_ROBIN_PERIOD, and MDMA transfers begin. The counter is decremented on every data transfer (as each data word is written to memory). After the transfer corresponding to a count of one, the MDMA stream selection is passed automatically to the other stream with zero overhead, and the MDMA_ROUND_ROBIN_COUNT counter is reloaded with the period value P from MDMA_ROUND_ROBIN_PERIOD. In this cycle, if the other MDMA stream is ready to perform a transfer, the stream selection is locked on the new MDMA stream. If the other MDMA stream is not ready to perform a transfer, then no transfer is performed, and the stream selection unlocks and becomes free again on the next cycle.

If round-robin operation is used when only one MDMA stream is active, one idle cycle will occur for each P MDMA data cycles, slightly lowering the bandwidth by a factor of 1/(P+1). However if both MDMA streams are used, memory DMA can operate continuously with zero additional overhead for alternation of streams. (Other than overhead cycles normally associated with reversal of read/write direction to memory). By selection of various round-robin period values P, which limit how often the MDMA streams alternate, maximal transfer efficiency can be maintained.

Traffic Control

In the Blackfin DMA architecture, there are two completely separate but simultaneous prioritization processes—the DAB bus prioritization and the memory bus (DCB and DEB) prioritization. Peripherals that are request-

ing DMA via the DAB bus, and whose data FIFOs are ready to handle the transfer, compete with each other for DAB bus cycles. Similarly but separately, channels whose FIFOs need memory service (prefetch or post-write) compete together for access to the memory buses. MDMA streams compete for memory access as a unit, and source and destination may be granted together if their memory transfers do not conflict. In this way, internal-to-external or external-to-internal memory transfers may occur at the full system clock rate (SCLK). Examples of memory conflict include simultaneous access to the same memory space and simultaneous attempts to fetch descriptors. Special processing may occur if a peripheral is requesting DMA but its FIFO is not ready (for example, an empty transmit FIFO or full receive FIFO). For more information, see "Temporary DMA Urgency" on page 6-46.

Traffic control is an important consideration in optimizing use of DMA resources. Traffic control is a way to influence how often the transfer direction on the data buses may change, by automatically grouping same direction transfers together. The DMA block provides a traffic control mechanism controlled by the DMA_TC_PER and DMA_TC_CNT registers. This mechanism performs the optimization without real-time processor intervention and without the need to program transfer bursts into the DMA work unit streams. Traffic can be independently controlled for each of the three buses (DAB, DCB, and DEB) with simple counters. In addition, alternation of transfers among MDMA streams can be controlled with the MDMA_ROUND_ROBIN_COUNT field of the DMA_TC_CNT register. See "Memory DMA Priority and Scheduling" on page 6-48.

Using the traffic control features, the DMA system preferentially grants data transfers on the DAB or memory buses which are going in the same read/write direction as the previous transfer, until either the traffic control counter times out or traffic stops or changes direction on its own. When the traffic counter reaches zero, the preference is changed to the opposite flow direction. These directional preferences work as if the priority of the opposite direction channels were decreased by 16.

For example, if channels 3 and 5 were requesting DAB access, but lower priority channel 5 is going with traffic and higher priority channel 3 is going against traffic, then channel 3's effective priority becomes 19, and channel 5 would be granted instead. If, on the next cycle, only channels 3 and 6 were requesting DAB transfers, and these transfer requests were both against traffic, then their effective priorities would become 19 and 22, respectively. One of the channels (channel 3) is granted, even though its direction is opposite to the current flow. No bus cycles are wasted, other than any necessary delay required for the bus turnaround.

This type of traffic control represents a trade-off of latency to improve utilization (efficiency). Higher traffic timeouts might increase the length of time each request waits for its grant, but it often dramatically improves the maximum attainable bandwidth in congested systems, often to above 90%.

To disable preferential DMA prioritization, program the `DMA_TC_PER` register to 0x0000.

Programming Model

Several synchronization and control methods are available for use in development of software tasks which manage peripheral DMA and memory DMA (see also "Memory DMA" on page 6-7). Such software needs to be able to accept requests for new DMA transfers from other software tasks, integrate these transfers into existing transfer queues, and reliably notify other tasks when the transfers are complete.

In the processor, it is possible for each peripheral DMA and memory DMA stream to be managed by a separate task or to be managed together with any other stream. Each DMA channel has independent, orthogonal control registers, resources, and interrupts, so that the selection of the control scheme for one channel does not affect the choice of control scheme on other channels. For example, one peripheral can use a linked-descriptor-list, interrupt-driven scheme while another peripheral

can simultaneously use a demand-driven, buffer-at-a-time scheme synchronized by polling of the DMAx_IRQ_STATUS register.

Synchronization of Software and DMA

A critical element of software DMA management is synchronization of DMA buffer completion with the software. This can best be done using interrupts, polling of DMAx_IRQ_STATUS, or a combination of both. Polling for address or count can only provide synchronization within loose tolerances comparable to pipeline lengths.

Interrupt-based synchronization methods must avoid interrupt overrun, or the failure to invoke a DMA channel's interrupt handler for every interrupt event due to excessive latency in processing of interrupts. Generally, the system design must either ensure that only one interrupt per channel is scheduled (for example, at the end of a descriptor list), or that interrupts are spaced sufficiently far apart in time that system processing budgets can guarantee every interrupt is serviced. Note, since every interrupt channel has its own distinct interrupt, interaction among the interrupts of different peripherals is much simpler to manage.

Due to DMA FIFOs and DMA/memory pipelining, polling of the DMAx_CURR_ADDR, DMAx_CURR_DESC_PTR, or DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT registers is not recommended for precisely synchronizing DMA with data processing. The current address, pointer, and count registers change several cycles in advance of the completion of the corresponding memory operation, as measured by the time at which the results of the operation would first be visible to the core by memory read or write instructions. For example, in a DMA memory write operation to external memory, assume a DMA write by channel A is initiated that causes the SDRAM to perform a page open operation which takes many system clock cycles. The DMA engine may then move on to another DMA operation by channel B which does not in itself incur latency, but will be stalled behind the slow operation of channel A. Software monitoring of channel B, based on examination of the DMAx_CURR_ADDR register contents,

would not safely conclude whether the memory location pointed to by channel B's DMAx_CURR_ADDR register has or has not been written.

If allowances are made for the lengths of the DMA/memory pipeline, polling of the current address, pointer, and count registers can permit loose synchronization of DMA with software. The depth of the DMA FIFO is four locations (either four 8- or 16-bit data elements, or two 32-bit data elements) for a peripheral DMA channel, and eight locations (four 32-bit data elements) for an MDMA FIFO. The DMA will not advance current address/pointer/count registers if these FIFOs are filled with incomplete work (including reads that have been started but not yet finished).

Additionally, the length of the combined DMA and L1 pipelines to internal memory is approximately six 8- or 16-bit data elements. The length of the DMA and external bus interface unit (EBIU) pipelines is approximately three data elements, when measured from the point where a DMA register update is visible to an MMR read to the point where DMA and core accesses to memory become strictly ordered. If the DMA FIFO length and the DMA/memory pipeline length are added, an estimate can be made of the maximum number of incomplete memory operations in progress at one time. This value is a maximum because the DMA/memory pipeline may include traffic from other DMA channels.

For example, assume a peripheral DMA channel is transferring a work unit of 100 data elements into internal memory and its DMAx_CURR_X_COUNT register reads a value of 60 remaining elements, so that processing of the first 40 elements has at least been started. Since the total pipeline length is no greater than the sum of four (for the peripheral DMA FIFO) plus six (for the DMA/memory pipeline) or ten data elements, it is safe to conclude that the DMA transfer of the first 30 (40-10) data elements is complete.

For precise synchronization, software should either wait for an interrupt or consult the channel's DMAx_IRQ_STATUS register to confirm completion of DMA, rather than polling current address/pointer/count registers. When the DMA system issues an interrupt or changes a DMAx_IRQ_STATUS

bit, it guarantees that the last memory operation of the work unit has been completed and will definitely be visible to processor code. For memory read DMA, the final memory read data will have been safely received in the DMA's FIFO. For memory write DMA, the DMA unit will have received an acknowledgement from L1 memory, or the EBIU, that the data has been written.

The following examples show methods of synchronizing software with several different styles of DMA.

Single-Buffer DMA Transfers

Synchronization is simple if a peripheral's DMA activity consists of isolated transfers of single buffers. DMA activity is initiated by software writes to the channel's control registers. The user may choose to use a single descriptor in memory, in which case the software only needs to write the DMAx_CONFIG and the DMAx_NEXT_DESC_PTR registers. Alternatively, the user may choose to write all the MMR registers directly from software, ending with the write to the DMAx_CONFIG register.

The simplest way to signal completion of DMA is by an interrupt. This is selected by the DI_EN bit in the DMAx_CONFIG register, and by the necessary setup of the system interrupt controller. If no interrupt is desired, the software can poll for completion by reading the DMAx_IRQ_STATUS register and testing the DMA_RUN bit. If this bit is zero, the buffer transfer has completed.

Continuous Transfers Using Autobuffering

If a peripheral's DMA data consists of a steady, periodic stream of signal data, DMA autobuffering (FLOW = 1) may be an effective option. Here, DMA is transferred from or to a memory buffer with a circular addressing

scheme, using either one- or two-dimensional indexing with zero processor and DMA overhead for looping. Synchronization options include:

- 1-D interrupt-driven—software is interrupted at the conclusion of each buffer. The critical design consideration is that the software must deal with the first items in the buffer before the next DMA transfer, which might overwrite or re-read the first buffer location before it is processed by software. This scheme may be workable if the system design guarantees that the data repeat period is longer than the interrupt latency under all circumstances.
- 2-D interrupt-driven (double buffering)—the DMA buffer is partitioned into two or more sub-buffers, and interrupts are selected (set DI_SEL = 1 in DMAx_CONFIG) to be signaled at the completion of each DMA inner loop. In this way, a traditional double buffer or "ping-pong" scheme can be implemented.

  For example, two 512-word sub-buffers inside a 1K-word buffer could be used to receive 16-bit peripheral data with these settings:

- DMAx_START_ADDR = buffer base address
- DMAx_CONFIG = 0x10D7 (FLOW = 1, DI_EN = 1, DI_SEL = 1, DMA2D = 1, WDSIZE = b#01, WNR = 1, DMAEN = 1)
- DMAx_X_COUNT = 512
- DMAx_X_MODIFY = 2 for 16-bit data
- DMAx_Y_COUNT = 2 for two sub-buffers
- DMAx_Y_MODIFY = 2 same as DMAx_X_MODIFY for contiguous sub-buffers
- 2-D polled—if interrupt overhead is unacceptable but the loose synchronization of address/count register polling is acceptable, a 2-D multibuffer synchronization scheme may be used. For exam-

ple, assume receive data needs to be processed in packets of sixteen 32-bit elements. A four-part 2-D DMA buffer can be allocated where each of the four sub-buffers can hold one packet with these settings:

- DMAx_START_ADDR = buffer base address
- DMAx_CONFIG = 0x101B (FLOW = 1, DI_EN = 0, DMA2D = 1, WDSIZE = b#10, WNR = 1, DMAEN = 1)
- DMAx_X_COUNT = 16
- DMAx_X_MODIFY = 4 for 32-bit data
- DMAx_Y_COUNT = 4 for four sub-buffers
- DMAx_Y_MODIFY = 4 same as DMAx_X_MODIFY for contiguous sub-buffers
- The synchronization core might read DMAx_Y_COUNT to determine which sub-buffer is currently being transferred, and then allow one full sub-buffer to account for pipelining. For example, if a read of DMAx_Y_COUNT shows a value of 3, then the software should assume that sub-buffer 3 is being transferred, but some portion of sub-buffer 2 may not yet be received. The software could, however, safely proceed with processing sub-buffers 1 or 0.
- 1-D unsynchronized FIFO—if a system's design guarantees that the processing of a peripheral's data and the DMA rate of the data will remain correlated in the steady state, but that short-term latency variations must be tolerated, it may be appropriate to build a simple FIFO. Here, the DMA channel may be programmed using 1-D autobuffer mode addressing without any interrupts or polling.

Descriptor Structures

DMA descriptors may be used to transfer data to or from memory data structures that are not simple 1-D or 2-D arrays. For example, if a packet of data is to be transmitted from several different locations in memory (a header from one location, a payload from a list of several blocks of memory managed by a memory pool allocator, and a small trailer containing a checksum), a separate DMA descriptor can be prepared for each memory area, and the descriptors can be grouped in either an array or list by selecting the appropriate FLOW setting in DMAx_CONFIG.

The software can synchronize with the progress of the structure's transfer by selecting interrupt notification for one or more of the descriptors. For example, the software might select interrupt notification for the header's descriptor and for the trailer's descriptor, but not for the payload blocks' descriptors.

It is important to remember the meaning of the various fields in the DMAx_CONFIG descriptor elements when building a list or array of DMA descriptors. In particular:

- The lower byte of DMAx_CONFIG specifies the DMA transfer to be performed by the *current* descriptor (for example 2-D interrupt-enable mode)
- The upper byte of DMAx_CONFIG specifies the format of the *next* descriptor in the chain. The NDSIZE and FLOW fields in a given descriptor do not correspond to the format of the descriptor itself; they specify the link to the next descriptor, if any.

On the other hand, when the DMA unit is being restarted, both bytes of the DMAx_CONFIG value written to the DMA channel's DMAx_CONFIG register should correspond to the current descriptor. At a minimum, the FLOW, NDSIZE, WNR, and DMAEN fields must all agree with the current descriptor. The WDSIZE, DI_EN, DI_SEL, SYNC, and DMA2D fields will be taken from the DMAx_CONFIG value in the descriptor read from memory. The field values initially written to the register are ignored. See "Initializing Descriptors in

Memory" on page 6-97 in the "Programming Examples" section for information on how descriptors can be set up.

Descriptor Queue Management

A system designer might want to write a DMA manager facility which accepts DMA requests from other software. The DMA manager software does not know in advance when new work requests will be received or what these requests might contain. The software could manage these transfers using a circular linked list of DMA descriptors, where each descriptor's NDPH and NDPL members point to the next descriptor, and the last descriptor points back to the first.

The code that writes into this descriptor list could use the processor's circular addressing modes (Ix, Lx, Mx, and Bx registers), so that it does not need to use comparison and conditional instructions to manage the circular structure. In this case, the NDPH and NDPL members of each descriptor could even be written once at startup and skipped over as each descriptor's new contents are written.

The recommended method for synchronization of a descriptor queue is through the use of an interrupt. The descriptor queue is structured so that at least the final valid descriptor is always programmed to generate an interrupt.

There are two general methods for managing a descriptor queue using interrupts:

- Interrupt on every descriptor
- Interrupt minimally - only on the last descriptor

Descriptor Queue Using Interrupts on Every Descriptor

In this system, the DMA manager software synchronizes with the DMA unit by enabling an interrupt on every descriptor. This method should

only be used if system design can guarantee that each interrupt event will be serviced separately (no interrupt overrun).

To maintain synchronization of the descriptor queue, the non-interrupt software maintains a count of descriptors added to the queue, while the interrupt handler maintains a count of completed descriptors removed from the queue. The counts are equal only when the DMA channel is paused after having processed all the descriptors.

When each new work request is received, the DMA manager software initializes a new descriptor, taking care to write a DMAx_CONFIG value with a FLOW value of 0. Next, the software compares the descriptor counts to determine if the DMA channel is running or not. If the DMA channel is paused (counts are equal), the software increments its count and then starts the DMA unit by writing the new descriptor's DMAx_CONFIG value to the DMA channel's DMAx_CONFIG register.

If the counts are unequal, the software instead modifies the next-to-last descriptor's DMAx_CONFIG value so that its upper half (FLOW and NDSIZE) now describes the newly queued descriptor. This operation does not disrupt the DMA channel, provided the rest of the descriptor data structure is initialized in advance. It is necessary, however, to synchronize the software to the DMA to correctly determine whether the new or the old DMAx_CONFIG value was read by the DMA channel.

This synchronization operation should be performed in the interrupt handler. First, upon interrupt, the handler should read the channel's DMAx_IRQ_STATUS register. If the DMA_RUN status bit is set, then the channel has moved on to processing another descriptor, and the interrupt handler may increment its count and exit. If the DMA_RUN status bit is not set, however, then the channel has paused, either because there are no more descriptors to process, or because the last descriptor was queued too late (the modification of the next-to-last descriptor's DMAx_CONFIG element occurred after that element was read into the DMA unit). In this case, the interrupt handler should write the DMAx_CONFIG value appropriate for the

last descriptor to the DMA channel's DMAx_CONFIG register, increment the completed descriptor count, and exit.

Again, this system can fail if the system's interrupt latencies are large enough to cause any of the channel's DMA interrupts to be dropped. An interrupt handler capable of safely synchronizing multiple descriptors' interrupts would need to be complex, performing several MMR accesses to ensure robust operation. In such a system environment, a minimal interrupt synchronization method is preferred.

Descriptor Queue Using Minimal Interrupts

In this system, only one DMA interrupt event is possible in the queue at any time. The DMA interrupt handler for this system can also be extremely short. Here, the descriptor queue is organized into an "active" and a "waiting" portion, where interrupts are enabled only on the last descriptor in each portion.

When each new DMA request is processed, the software's non-interrupt code fills in a new descriptor's contents and adds it to the waiting portion of the queue. The descriptor's DMAx_CONFIG word should have a FLOW value of zero. If more than one request is received before the DMA queue completion interrupt occurs, the non-interrupt code should queue later descriptors, forming a waiting portion of the queue that is disconnected from the active portion of the queue being processed by the DMA unit. In other words, all but the last active descriptors contain FLOW values ≥ 4 and have no interrupt enable set, while the last active descriptor contains a FLOW of 0 and an interrupt enable bit DI_EN set to 1. Also, all but the last waiting descriptors contain FLOW values ≥ 4 and no interrupt enables set, while the last waiting descriptor contains a FLOW of 0 and an interrupt enable bit set. This ensures that the DMA unit can automatically process the whole active queue and then issue one interrupt. Also, this arrangement makes it easy to start the waiting queue within the interrupt handler with a single DMAx_CONFIG register write.

After queuing a new waiting descriptor, the non-interrupt software should leave a message for its interrupt handler in a memory mailbox location containing the desired DMAx_CONFIG value to use to start the first waiting descriptor in the waiting queue (or 0 to indicate no descriptors are waiting).

Once processing by the DMA unit has started, it is critical that the software not directly modify the contents of the active descriptor queue unless careful synchronization measures are taken. In the most straightforward implementation of a descriptor queue, the DMA manager software would never modify descriptors on the active queue; instead, the DMA manager waits until the DMA queue completion interrupt indicates the processing of the entire active queue is complete.

When a DMA queue completion interrupt is received, the interrupt handler reads the mailbox from the non-interrupt software and writes the value in it to the DMA channel's DMAx_CONFIG register. This single register write restarts the queue, effectively transforming the waiting queue to an active queue. The interrupt handler should then pass a message back to the non-interrupt software indicating the location of the last descriptor accepted into the active queue. If, on the other hand, the interrupt handler reads its mailbox and finds a DMAx_CONFIG value of zero, indicating there is no more work to perform, then it should pass an appropriate message (for example zero) back to the non-interrupt software indicating that the queue has stopped. This simple handler should be able to be coded in a very small number of instructions.

The non-interrupt software which accepts new DMA work requests needs to synchronize the activation of new work with the interrupt handler. If the queue has stopped (the mailbox from the interrupt software is zero), the non-interrupt software is responsible for starting the queue (writing the first descriptor's DMAx_CONFIG value to the channel's DMAx_CONFIG register). If the queue is not stopped, the non-interrupt software must not write to the DMAx_CONFIG register (which would cause a DMA error).

Instead the descriptor should queue to the waiting queue, and update its mailbox directed to the interrupt handler.

Software Triggered Descriptor Fetches

If a DMA has been stopped in FLOW = 0 mode, the DMA_RUN bit in the DMAx_IRQ_STATUS register remains set until the content of the internal DMA FIFOs has been completely processed. Once the DMA_RUN bit clears, it is safe to restart the DMA by simply writing again to the DMAx_CONFIG register. The DMA sequence is repeated with the previous settings.

Similarly, a descriptor-based DMA sequence that has been stopped temporarily with a FLOW = 0 descriptor can be continued with a new write to the configuration register. When the DMA controller detects the FLOW = 0 condition by loading the DMACFG field from memory, it has already updated the next descriptor pointer, regardless of whether operating in descriptor array mode or descriptor list mode.

The next descriptor pointer remains valid if the DMA halts and is restarted. As soon as the DMA_RUN bit clears, software can restart the DMA and force the DMA controller to fetch the next descriptor. To accomplish this, the software writes a value with the DMAEN bit set and with proper values in the FLOW and NDSIZE fields into the configuration register. The next descriptor is fetched if FLOW equals 0x4, 0x6, or 0x7. In this mode of operation, the NDSIZE field should at least span up to the DMACFG field to overwrite the configuration register immediately.

One possible procedure is:

1. Write to DMAx_NEXT_DESC_PTR
2. Write to DMAx_CONFIG with
    - FLOW = 0x8
    - NDSIZE ≥ 0xA

- DI_EN = 0
- DMAEN = 1

3. Automatically fetched DMACFG has
    - FLOW = 0x0
    - NDSIZE = 0x0
    - SYNC = 1 (for transmitting DMAs only)
    - DI_EN = 1
    - DMAEN = 1
4. In the interrupt routine, repeat step 2. The DMAx_NEXT_DESC_PTR is updated by the descriptor fetch.

To avoid polling of the DMA_RUN bit, set the SYNC bit in case of memory read DMAs (DMA transmit or MDMA source).

If all DMACFG fields in a descriptor chain have the FLOW and NDSIZE fields set to zero, the individual DMA sequences do not start until triggered by software. This is useful when the DMAs need to be synchronized with other events in the system, and it is typically performed by interrupt service routines. A single MMR write is required to trigger the next DMA sequence.

Especially when applied to MDMA channels, such scenarios play an important role. Usually, the timing of MDMAs cannot be controlled (See "Handshaked Memory DMA Operation" on page 6-107). By halting descriptor chains or rings this way, the whole DMA transaction can be broken into pieces that are individually triggered by software.

Source and destination channels of a MDMA may differ in descriptor structure. However, the total work count must match when the DMA stops. Whenever a MDMA is stopped, destination and

source channels should both provide the same FLOW = 0 mode after exactly the same number of words. Accordingly, both channels need to be started afterward.

Software-triggered descriptor fetches are illustrated in Listing 6-7 on page 6-100. MDMA channels can be paused by software at any time by writing a 0 to the DRQ bit field in the HMDMAx_CONTROL register. This simply disables the self-generated DMA requests, whether or not the HMDMA is enabled.

DMA Registers

DMA registers fall into three categories:

- DMA channel registers
- Handshaked MDMA registers
- Global DMA traffic control registers

DMA Channel Registers

A processor features up to twelve peripheral DMA channels and two channel pairs for memory DMA. All channels have an identical set of registers as summarized in Table 6-4.

Table 6-4 lists the generic names of the DMA registers. For each register, the table also shows the MMR offset, a brief description of the register,

the register category, and where applicable, the corresponding name for the data element in a DMA descriptor.

Table 6-4. Generic Names of the DMA Memory-mapped Registers

MMR Offset	Generic MMR Name	MMR Description	Register Category	Name of Corresponding Descriptor Element in Memory
0x00	NEXT_DESC_PTR	Link pointer to next descriptor	Parameter	NDPH (upper 16 bits), NDPL (lower 16 bits)
0x04	START_ADDR	Start address of current buffer	Parameter	SAH (upper 16 bits), SAL (lower 16 bits)
0x08	CONFIG	DMA Configuration register, including enable bit	Parameter	DMACFG
0x0C	Reserved	Reserved		
0x10	X_COUNT	Inner loop count	Parameter	XCNT
0x14	X_MODIFY	Inner loop address increment, in bytes	Parameter	XMOD
0x18	Y_COUNT	Outer loop count (2-D only)	Parameter	YCNT
0x1C	Y_MODIFY	Outer loop address increment, in bytes	Parameter	YMOD
0x20	CURR_DESC_PTR	Current descriptor pointer	Current	N/A
0x24	CURR_ADDR	Current DMA address	Current	N/A
0x28	IRQ_STATUS	Interrupt status register contains completion and DMA error interrupt status and channel state (run/fetch/paused)	Control/ Status	N/A

Table 6-4. Generic Names of the DMA Memory-mapped Registers (Continued)

MMR Offset	Generic MMR Name	MMR Description	Register Category	Name of Corresponding Descriptor Element in Memory
0x2C	PERIPHERAL_MAP	Peripheral to DMA channel mapping contains a 4-bit value specifying the peripheral associated with this DMA channel (read-only for MDMA channels)	Control/ Status	N/A
0x30	CURR_X_COUNT	Current count (1-D) or intra-row X count (2-D); counts down from X_COUNT	Current	N/A
0x34	Reserved	Reserved		
0x38	CURR_Y_COUNT	Current row count (2-D only); counts down from Y_COUNT	Current	N/A
0x3C	Reserved	Reserved		

Channel-specific register names are composed of a prefix and the generic MMR name shown in Table 6-4. For peripheral DMA channels the prefix "DMAx_" is used, where "x" stands for a channel number between 0 and 11. For memory DMA channels, the prefix is "MDMA_yy_", where "yy" stands for either "D0", "S0", "D1", or "S1" to indicate destination and source channel registers of MDMA0 and MDMA1. For example the peripheral DMA channel 6 configuration register is called DMA6_CONFIG. The register for the MDMA1 source channel is called MDMA_S1_CONFIG.

The generic MMR names shown in Table 6-4 are not actually mapped to resources in the processor.

For convenience, discussions in this chapter use generic (non-peripheral specific) DMA and memory DMA register names.

DMA channel registers fall into three categories.

- Parameter registers such as DMAx_CONFIG and DMAx_X_COUNT that can be loaded directly from descriptor elements as shown in Table 6-4
- Current registers such as DMAx_CURR_ADDR and DMAx_CURR_X_COUNT
- Control/status registers such as DMAx_IRQ_STATUS and DMAx_PERIPHERAL_MAP

All DMA registers can be accessed as 16-bit entities. However, the following registers may also be accessed as 32-bit registers.

- DMAx_NEXT_DESC_PTR
- DMAx_START_ADDR
- DMAx_CURR_DESC_PTR
- DMAx_CURR_ADDR

When these four registers are accessed as 16-bit entities, only the lower 16 bits can be accessed.

Because confusion might arise between descriptor element names and generic DMA register names, this chapter uses different naming conventions for physical registers and their corresponding elements in descriptors that reside in memory. Table 6-4 shows the relation.

DMA Peripheral Map Registers(DMAx_PERIPHERAL_MAP/ MDMA_yy_PERIPHERAL_MAP)

Each DMA channel's DMAx_PERIPHERAL_MAP register contains bits that:

- Map the channel to a specific peripheral
- Identify whether the channel is a peripheral DMA channel or a memory DMA channel

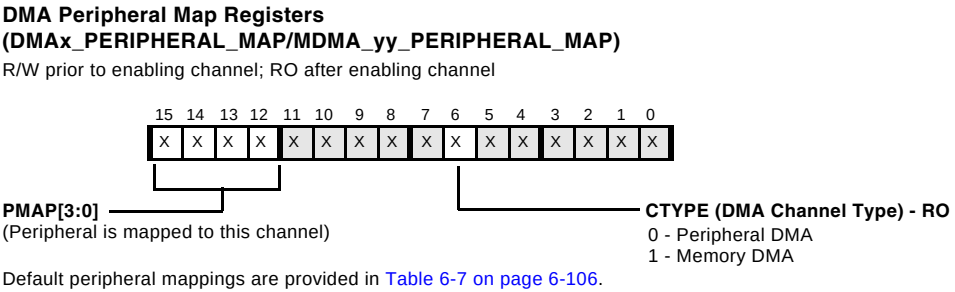

DMA Peripheral Map Registers
(DMAx_PERIPHERAL_MAP/MDMA_yy_PERIPHERAL_MAP)
R/W prior to enabling channel; RO after enabling channel
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
PMAP[3:0]
(Peripheral is mapped to this channel)
CTYPE (DMA Channel Type) - RO
0 - Peripheral DMA
1 - Memory DMA
Default peripheral mappings are provided in Table 6-7 on page 6-106.

Figure 6-5. DMA Peripheral Map Registers

Follow these steps to swap the DMA channel priorities of two channels. Assume that channels 6 and 7 are involved.

1. Make sure DMA is disabled on channels 6 and 7.
2. Write DMA6_PERIPHERAL_MAP with 0x7000 and DMA7_PERIPHERAL_MAP with 0x6000.
3. Enable DMA on channels 6 and/or 7.

DMA Configuration Registers (DMAx_CONFIG/MDMA_yy_CONFIG)

The DMAx_CONFIG register, shown in Figure 6-6, is used to set up DMA parameters and operating modes. Writing the DMAx_CONFIG register while

DMA is already running will cause a DMA error unless writing with the DMAEN bit set to 0.

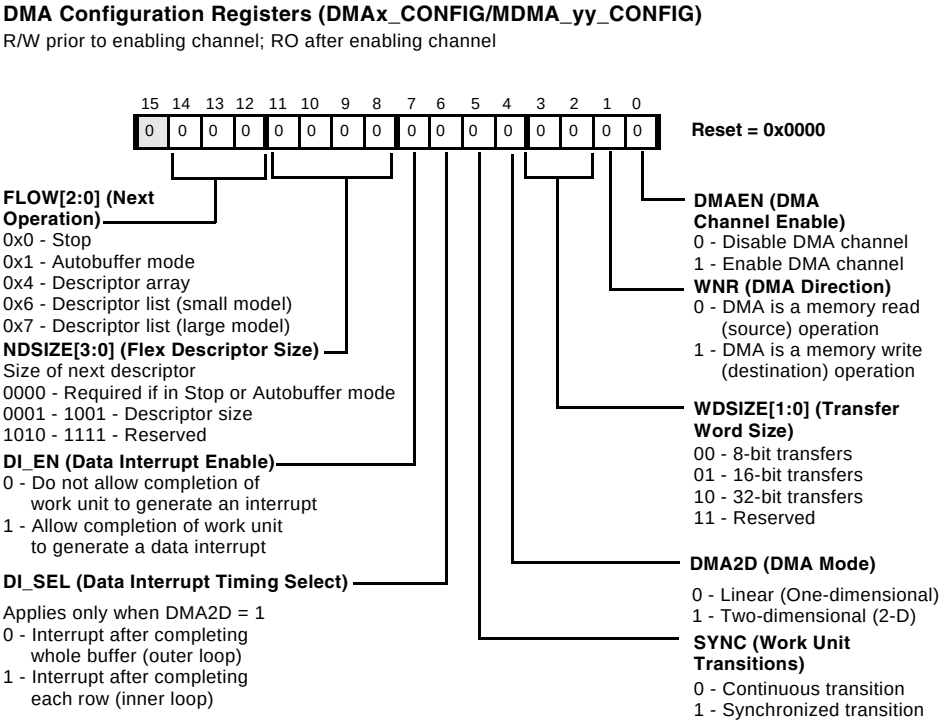

DMA Configuration Registers (DMAx_CONFIG/MDMA_yy_CONFIG)
R/W prior to enabling channel; RO after enabling channel
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
FLOW[2:0] (Next Operation)
0x0 - Stop
0x1 - Autobuffer mode
0x4 - Descriptor array
0x6 - Descriptor list (small model)
0x7 - Descriptor list (large model)
NDSIZE[3:0] (Flex Descriptor Size)
Size of next descriptor
0000 - Required if in Stop or Autobuffer mode
0001 - 1001 - Descriptor size
1010 - 1111 - Reserved
DI_EN (Data Interrupt Enable)
0 - Do not allow completion of work unit to generate an interrupt
1 - Allow completion of work unit to generate a data interrupt
DI_SEL (Data Interrupt Timing Select)
Applies only when DMA2D = 1
0 - Interrupt after completing whole buffer (outer loop)
1 - Interrupt after completing each row (inner loop)
DMAEN (DMA Channel Enable)
0 - Disable DMA channel
1 - Enable DMA channel
WNR (DMA Direction)
0 - DMA is a memory read (source) operation
1 - DMA is a memory write (destination) operation
WDSIZE[1:0] (Transfer Word Size)
00 - 8-bit transfers
01 - 16-bit transfers
10 - 32-bit transfers
11 - Reserved
DMA2D (DMA Mode)
0 - Linear (One-dimensional)
1 - Two-dimensional (2-D)
SYNC (Work Unit Transitions)
0 - Continuous transition
1 - Synchronized transition

Figure 6-6. DMA Configuration Registers

The fields of the DMAx_CONFIG register are used to set up DMA parameters and operating modes.

- FLOW[2:0] (next operation). This field specifies the type of DMA transfer to follow the present one. The flow options are:
- 0x0 - stop. When the current work unit completes, the DMA channel stops automatically, after signaling an interrupt (if selected). The DMA_RUN status bit in the DMAx_IRQ_STATUS register changes

from 1 to 0, while the DMAEN bit in the DMAx_CONFIG register is unchanged. In this state, the channel is paused. Peripheral interrupts are still filtered out by the DMA unit. The channel may be restarted simply by another write to the DMAx_CONFIG register specifying the next work unit, in which the DMAEN bit is set to 1.

0x1 - autobuffer mode. In this mode, no descriptors in memory are used. Instead, DMA is performed in a continuous circular buffer fashion based on user-programmed DMA MMR settings. Upon completion of the work unit, the parameter registers are reloaded into the current registers, and DMA resumes immediately with zero overhead. Autobuffer mode is stopped by a user write of 0 to the DMAEN bit in the DMAx_CONFIG register.

0x4 - descriptor array mode. This mode fetches a descriptor from memory that does not include the NDPH or NDPL elements. Because the descriptor does not contain a next descriptor pointer entry, the DMA engine defaults to using the DMAx_CURR_DESC_PTR register to step through descriptors, thus allowing a group of descriptors to follow one another in memory like an array.

0x6 - descriptor list (small model) mode. This mode fetches a descriptor from memory that includes NDPL, but not NDPH. Therefore, the high 16 bits of the next descriptor pointer field are taken from the upper 16 bits of the DMAx_NEXT_DESC_PTR register, thus confining all descriptors to a specific 64K page in memory.

0x7 - descriptor list (large model) mode. This mode fetches a descriptor from memory that includes NDPH and NDPL, thus allowing maximum flexibility in locating descriptors in memory.

- NDSIZE[3:0] (flex descriptor size). This field specifies the number of descriptor elements in memory to load. This field must be 0 if in stop or autobuffer mode. If NDSIZE and FLOW specify a descriptor that extends beyond YMOD, a DMA error results.
- DI_EN (data interrupt enable). This bit specifies whether to allow completion of a work unit to generate a data interrupt.
- DI_SEL (data interrupt timing select). This bit specifies the timing of a data interrupt—after completing the whole buffer or after completing each row of the inner loop. This bit is used only in 2-D DMA operation.
- SYNC (work unit transitions). This bit specifies whether the DMA channel performs a continuous transition (SYNC = 0) or a synchronized transition (SYNC = 1) between work units. For more information, see "Work Unit Transitions" on page 6-25.

  In DMA transmit (memory read) and MDMA source channels, the SYNC bit controls the interrupt timing at the end of the work unit and the handling of the DMA FIFO between the current and next work unit.

Work unit transitions for MDMA streams are controlled by the SYNC bit of the MDMA source channel's DMAx_CONFIG register. The SYNC bit of the MDMA destination channel is reserved and must be 0.

- DMA2D (DMA mode). This bit specifies whether DMA mode involves only DMAx_X_COUNT and DMAx_X_MODIFY (one-dimensional DMA) or also involves DMAx_Y_COUNT and DMAx_Y_MODIFY (two-dimensional DMA).
- WDSIZE[1:0] (transfer word size). The DMA engine supports transfers of 8-, 16-, or 32-bit items. Each request/grant results in a single memory access (although two cycles are required to transfer 32-bit data through a 16-bit memory port or through the 16-bit DMA access bus). The increment sizes (strides) of the DMA address pointer registers must be a multiple of the transfer unit size—one for 8-bit, two for 16-bit, four for 32-bit.

  Only SPORT DMA and Memory DMA can operate with a transfer size of 32 bits. All other peripherals have a maximum DMA transfer size of 16 bits.
- WNR (DMA direction). This bit specifies DMA direction—memory read (0) or memory write (1).
- DMAEN (DMA channel enable). This bit specifies whether to enable a given DMA channel.

When a peripheral DMA channel is enabled, interrupts from the peripheral denote DMA requests. When a channel is disabled, the DMA unit ignores the peripheral interrupt and passes it directly to the interrupt controller. To avoid unexpected results, take care to enable the DMA channel before enabling the peripheral, and to disable the peripheral before disabling the DMA channel.

DMA Interrupt Status Registers (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS)

The DMAx_IRQ_STATUS register, shown in Figure 6-7, contains bits that record whether the DMA channel:

- Is enabled and operating, enabled but stopped, or disabled.
- Is fetching data or a DMA descriptor.
- Has detected that a global DMA interrupt or a channel interrupt is being asserted.
- Has logged occurrence of a DMA error.

Note the DMA_DONE interrupt is asserted when the last memory access (read or write) has completed.

For a memory transfer to a peripheral, there may be up to four data words in the channel's DMA FIFO when the interrupt occurs. At this point, it is normal to immediately start the next work unit. If, however, the application needs to know when the final data item is

actually transferred to the peripheral, the application can test or poll the DMA_RUN bit. As long as there is undelivered transmit data in the FIFO, the DMA_RUN bit is 1.

For a memory write DMA channel, the state of the DMA_RUN bit has no meaning after the last DMA_DONE event has been signaled. It does not indicate the status of the DMA FIFO.

For MDMA transfers where an interrupt is not desired to notify when the DMA operation has ended, software should poll the DMA_DONE bit, rather than the DMA_RUN bit to determine when the transaction has completed.

DMA Interrupt Status Registers (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS)

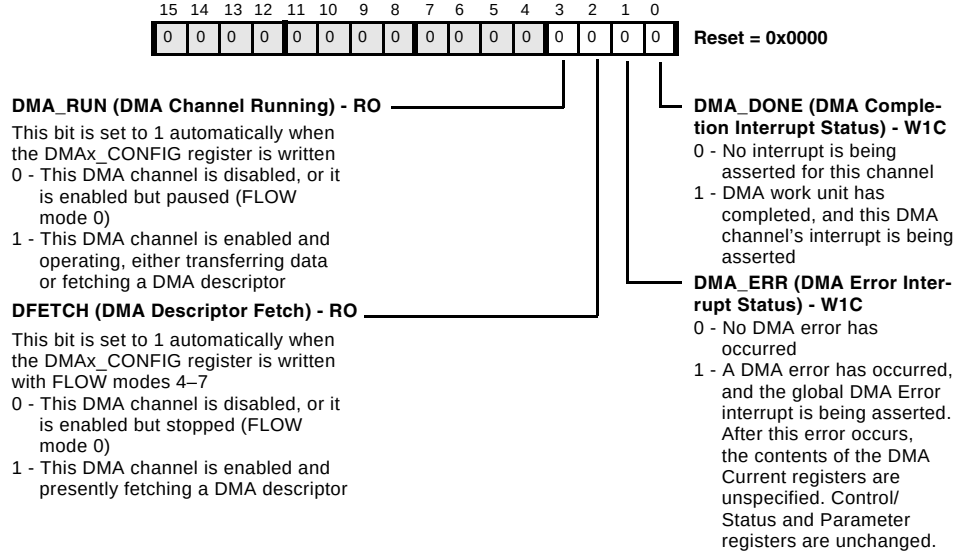

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DMA_RUN (DMA Channel Running) - RO
This bit is set to 1 automatically when the DMAx_CONFIG register is written
0 - This DMA channel is disabled, or it is enabled but paused (FLOW mode 0)
1 - This DMA channel is enabled and operating, either transferring data or fetching a DMA descriptor
DFETCH (DMA Descriptor Fetch) - RO
This bit is set to 1 automatically when the DMAx_CONFIG register is written with FLOW modes 4–7
0 - This DMA channel is disabled, or it is enabled but stopped (FLOW mode 0)
1 - This DMA channel is enabled and presently fetching a DMA descriptor
DMA_DONE (DMA Completion Interrupt Status) - W1C
0 - No interrupt is being asserted for this channel
1 - DMA work unit has completed, and this DMA channel's interrupt is being asserted
DMA_ERR (DMA Error Interrupt Status) - W1C
0 - No DMA error has occurred
1 - A DMA error has occurred, and the global DMA Error interrupt is being asserted. After this error occurs, the contents of the DMA Current registers are unspecified. Control/ Status and Parameter registers are unchanged.

Figure 6-7. DMA Interrupt Status Registers

The processor supports a flexible interrupt control structure with three interrupt sources:

- Data driven interrupts (see Table 6-5)
- Peripheral error interrupts
- DMA error interrupts (for example, bad descriptor or bus error)

Separate interrupt request (IRQ) levels are allocated for data, peripheral error, and DMA error interrupts.

Table 6-5. Data Driven Interrupts

Interrupt Name	Description
No Interrupt	Interrupts can be disabled for a given work unit.
Peripheral Interrupt	These are peripheral (non-DMA) interrupts.
Row Completion	DMA Interrupts can occur on the completion of a row (CURR_X_COUNT expiration).
Buffer Completion	DMA Interrupts can occur on the completion of an entire buffer (when CURR_X_COUNT and CURR_Y_COUNT expire).

The DMA error conditions for all DMA channels are OR'd together into one system-level DMA error interrupt. The individual IRQ_STATUS words of each channel can be read to identify the channel that caused the DMA error interrupt.

Note the DMA_DONE and DMA_ERR interrupt indicators are write-one-to-clear (W1C).

When switching a peripheral from DMA to non-DMA mode, the peripheral's interrupts should be disabled during the mode switch (via the appropriate peripheral register or SIC_IMASK register) so that no unintended interrupt is generated on the shared DMA/interrupt request line.

DMA Start Address Registers (DMAx_START_ADDR/MDMA_yy_START_ADDR)

The DMAx_START_ADDR register, shown in Figure 6-8, contains the start address of the data buffer currently targeted for DMA.

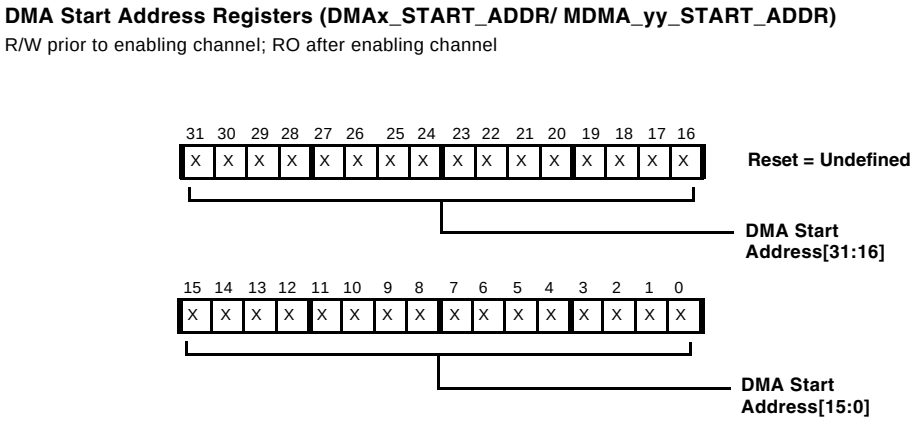

DMA Start Address Registers (DMAx_START_ADDR/ MDMA_yy_START_ADDR)
R/W prior to enabling channel; RO after enabling channel
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
DMA Start Address[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
DMA Start Address[15:0]

Figure 6-8. DMA Start Address Registers

DMA Current Address Registers (DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR)

The 32-bit DMAx_CURR_ADDR register shown in Figure 6-9, contains the present DMA transfer address for a given DMA session. On the first memory transfer of a DMA work unit, the DMAx_CURR_ADDR register is loaded

from the DMAx_START_ADDR register, and it is incremented as each transfer occurs.

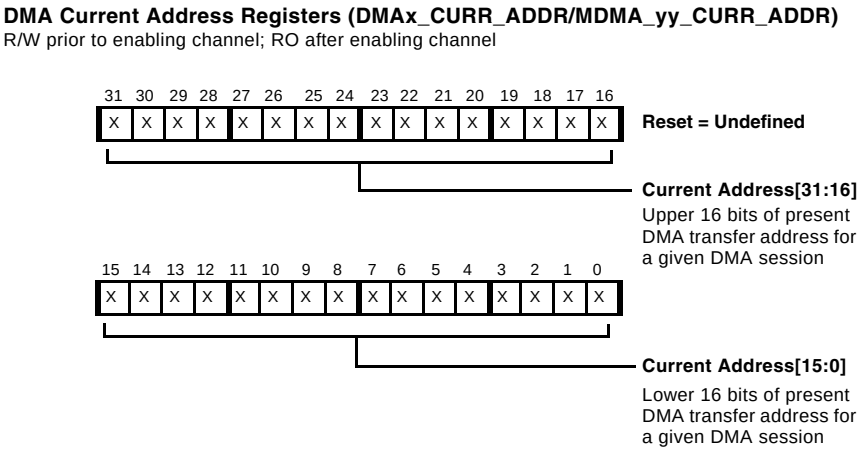

DMA Current Address Registers (DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR)
R/W prior to enabling channel; RO after enabling channel
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Current Address[31:16]
Upper 16 bits of present DMA transfer address for a given DMA session
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Current Address[15:0]
Lower 16 bits of present DMA transfer address for a given DMA session

Figure 6-9. DMA Current Address Registers

DMA Inner Loop Count Registers (DMAx_X_COUNT/MDMA_yy_X_COUNT)

For 2-D DMA, the DMAx_X_COUNT register, shown in Figure 6-10, contains the inner loop count. For 1-D DMA, it specifies the number of elements

to transfer. For details, see "Two-Dimensional DMA Operation" on page 6-12. A value of 0 in DMAx_X_COUNT corresponds to 65,536 elements.

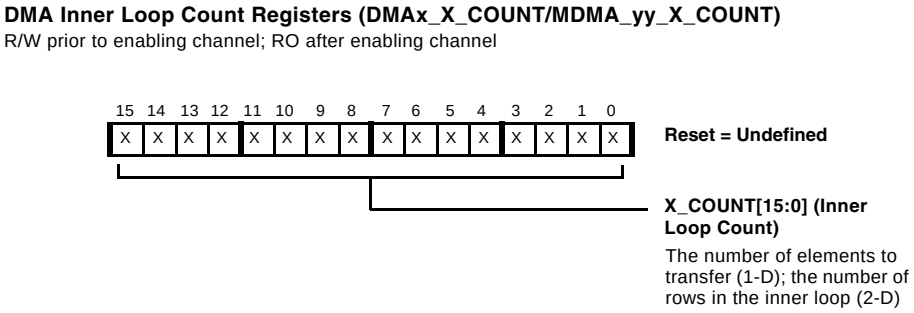

DMA Inner Loop Count Registers (DMAx_X_COUNT/MDMA_yy_X_COUNT)
R/W prior to enabling channel; RO after enabling channel
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Reset = Undefined
X_COUNT[15:0] (Inner Loop Count)
The number of elements to transfer (1-D); the number of rows in the inner loop (2-D)

Figure 6-10. DMA Inner Loop Count Registers

DMA Current Inner Loop Count Registers (DMAx_CURR_X_COUNT /MDMA_yy_CURR_X_COUNT)

The DMAx_CURR_X_COUNT register, shown in Figure 6-11, holds the number of transfers remaining in the current DMA row (inner loop). On the first memory transfer of each DMA work unit, it is loaded with the value in the DMAx_X_COUNT register and then decremented. For 2-D DMA, on the last memory transfer in each row except the last row, it is reloaded with the value in the DMAx_X_COUNT register; this occurs at the same time that the value in the DMAx_CURR_Y_COUNT register is decremented. Otherwise it is decremented each time an element is transferred. Expiration of the count in this register signifies that DMA is complete. In 2-D DMA, the DMAx_CURR_X_COUNT register value is 0 only when the entire transfer is

complete. Between rows it is equal to the value of the DMAx_X_COUNT register.

DMA Current Inner Loop Count Registers (DMAx_CURR_X_COUNT/ MDMA_yy_CURR_X_COUNT)

R/W prior to enabling channel; RO after enabling channel

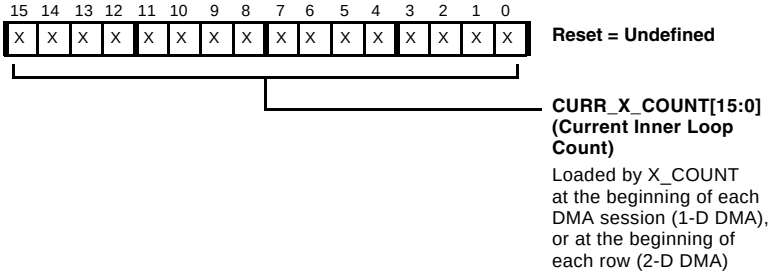

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Reset = Undefined
CURR_X_COUNT[15:0] (Current Inner Loop Count)
Loaded by X_COUNT at the beginning of each DMA session (1-D DMA), or at the beginning of each row (2-D DMA)

Figure 6-11. DMA Current Inner Loop Count Registers

DMA Inner Loop Address Increment Registers (DMAx_X_MODIFY/MDMA_yy_X_MODIFY)

The DMAx_X_MODIFY register, shown in Figure 6-12, contains a signed, two's-complement byte-address increment. In 1-D DMA, this increment is the stride that is applied after transferring each element.

DMAx_X_MODIFY is specified in bytes, regardless of the DMA transfer size.

In 2-D DMA, this increment is applied after transferring each element in the inner loop, up to but not including the last element in each inner loop. After the last element in each inner loop, the DMAx_Y_MODIFY register is applied instead, except on the very last transfer of each work unit. The DMAx_X_MODIFY register is always applied to the last transfer of a work unit.

The DMAx_X_MODIFY field may be set to 0. In this case, DMA is performed repeatedly to or from the same address. This is useful, for example, in

transferring data between a data register and an external memory-mapped peripheral.

DMA Inner Loop Address Increment Registers (DMAx_X_MODIFY/MDMA_yy_X_MODIFY)
R/W prior to enabling channel; RO after enabling channel

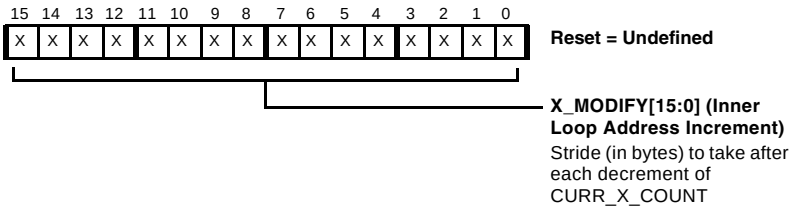

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Reset = Undefined
X_MODIFY[15:0] (Inner Loop Address Increment)
Stride (in bytes) to take after each decrement of CURR_X_COUNT

Figure 6-12. DMA Inner Loop Address Increment Registers

DMA Outer Loop Count Registers (DMAx_Y_COUNT/MDMA_yy_Y_COUNT)

For 2-D DMA, the DMAx_Y_COUNT register, shown in Figure 6-13, contains the outer loop count. It is not used in 1-D DMA mode. This register contains the number of rows in the outer loop of a 2-D DMA sequence. For details, see "Two-Dimensional DMA Operation" on page 6-12.

DMA Outer Loop Count Registers (DMAx_Y_COUNT/MDMA_yy_Y_COUNT)
R/W prior to enabling channel; RO after enabling channel

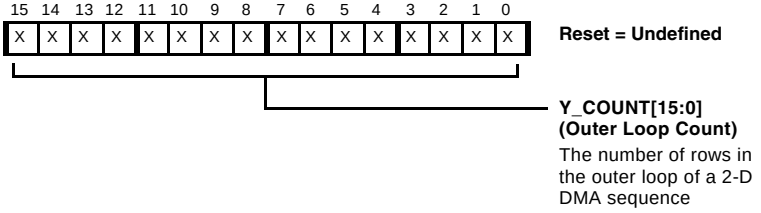

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Reset = Undefined
Y_COUNT[15:0] (Outer Loop Count)
The number of rows in the outer loop of a 2-D DMA sequence

Figure 6-13. DMA Outer Loop Count Registers

DMA Current Outer Loop Count Registers (DMAx_CURR_Y_COUNT/ MDMA_yy_CURR_Y_COUNT)

The DMAx_CURR_Y_COUNT register, used only in 2-D mode, holds the number of full or partial rows (outer loops) remaining in the current work unit. See Figure 6-14. On the first memory transfer of each DMA work unit, it is loaded with the value of the DMAx_Y_COUNT register. The register is decremented each time the DMAx_CURR_X_COUNT register expires during 2-D DMA operation (1 to DMAx_X_COUNT or 1 to 0 transition), signifying completion of an entire row transfer. After a 2-D DMA session is complete, DMAx_CURR_Y_COUNT = 1 and DMAx_CURR_X_COUNT = 0.

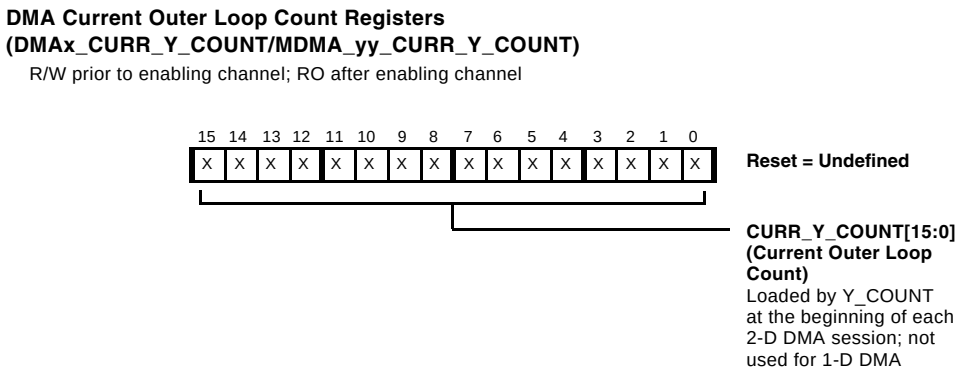

DMA Current Outer Loop Count Registers
(DMAx_CURR_Y_COUNT/MDMA_yy_CURR_Y_COUNT)
R/W prior to enabling channel; RO after enabling channel
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Reset = Undefined
CURR_Y_COUNT[15:0]
(Current Outer Loop
Count)
Loaded by Y_COUNT
at the beginning of each
2-D DMA session; not
used for 1-D DMA

Figure 6-14. DMA Current Outer Loop Count Registers

DMA Outer Loop Address Increment Registers (DMAx_Y_MODIFY/MDMA_yy_Y_MODIFY)

The DMAx_Y_MODIFY register contains a signed, two's-complement value. See Figure 6-15. This byte-address increment is applied after each decrement of the DMAx_CURR_Y_COUNT register except for the last item in the 2-D array where the DMAx_CURR_Y_COUNT also expires. The value is the offset

between the last word of one row and the first word of the next row. For details, see "Two-Dimensional DMA Operation" on page 6-12.

DMAx_Y_MODIFY is specified in bytes, regardless of the DMA transfer size.

DMA Outer Loop Address Increment Registers (DMAx_Y_MODIFY/ MDMA_yy_Y_MODIFY)
R/W prior to enabling channel; RO after enabling channel

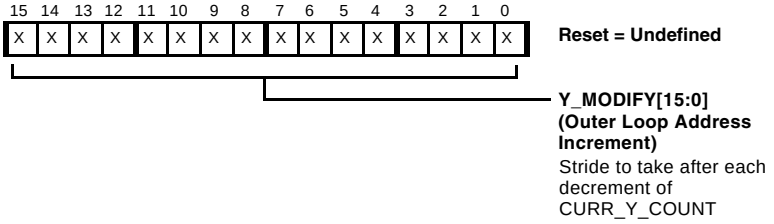

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Reset = Undefined
Y_MODIFY[15:0]
(Outer Loop Address Increment)
Stride to take after each decrement of CURR_Y_COUNT

Figure 6-15. DMA Outer Loop Address Increment Registers

DMA Next Descriptor Pointer Registers (DMAx_NEXT_DESC_PTR/ MDMA_yy_NEXT_DESC_PTR)

The 32-bit DMAx_NEXT_DESC_PTR register, shown in Figure 6-16, specifies where to look for the start of the next descriptor block when the DMA activity specified by the current descriptor block finishes. This register is used in small and large descriptor list modes. At the start of a descriptor fetch in either of these modes, this register is copied into the DMAx_CURR_DESC_PTR register. Then, during the descriptor fetch, the DMAx_CURR_DESC_PTR register increments after each element of the descriptor is read in.

In small and large descriptor list modes, the DMAx_NEXT_DESC_PTR register, and not the DMAx_CURR_DESC_PTR register, must be programmed directly via MMR access before starting DMA operation.

In descriptor array mode, the next descriptor pointer register is disregarded, and fetching is controlled only by the DMAx_CURR_DESC_PTR register.

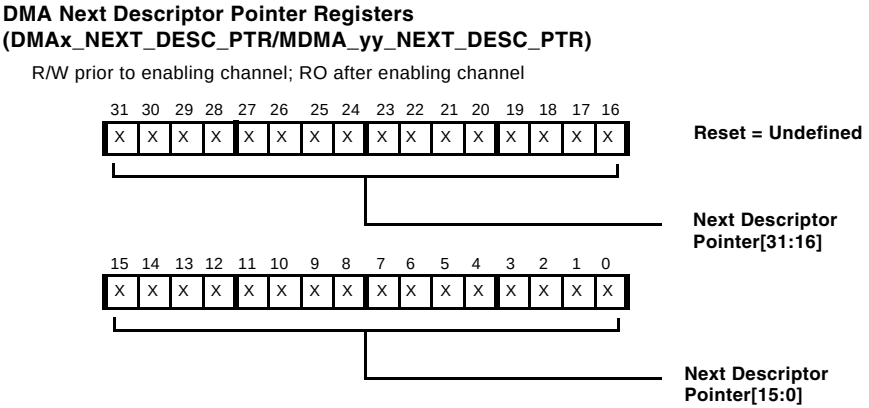

DMA Next Descriptor Pointer Registers
(DMAx_NEXT_DESC_PTR/MDMA_yy_NEXT_DESC_PTR)
R/W prior to enabling channel; RO after enabling channel
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Next Descriptor
Pointer[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Next Descriptor
Pointer[15:0]

Figure 6-16. DMA Next Descriptor Pointer Registers

DMA Current Descriptor Pointer Registers (DMAx_CURR_DESC_PTR/ MDMA_yy_CURR_DESC_PTR)

The 32-bit DMAx_CURR_DESC_PTR register, shown in Figure 6-17, contains the memory address for the next descriptor element to be loaded. For FLOW mode settings that involve descriptors (FLOW = 4, 6, or 7), this register is used to read descriptor elements into appropriate MMRs before a DMA work block begins. For descriptor list modes (FLOW = 6 or 7), this register is initialized from the DMAx_NEXT_DESC_PTR register before loading each descriptor. Then, the address in the DMAx_CURR_DESC_PTR register increments as each descriptor element is read in.

When the entire descriptor has been read, the DMAx_CURR_DESC_PTR register contains this value:

Descriptor Start Address + (2 × Descriptor Size) (# of elements)

For descriptor array mode (FLOW = 4), this register, and not the DMAx_NEXT_DESC_PTR register, must be programmed by MMR access before starting DMA operation.

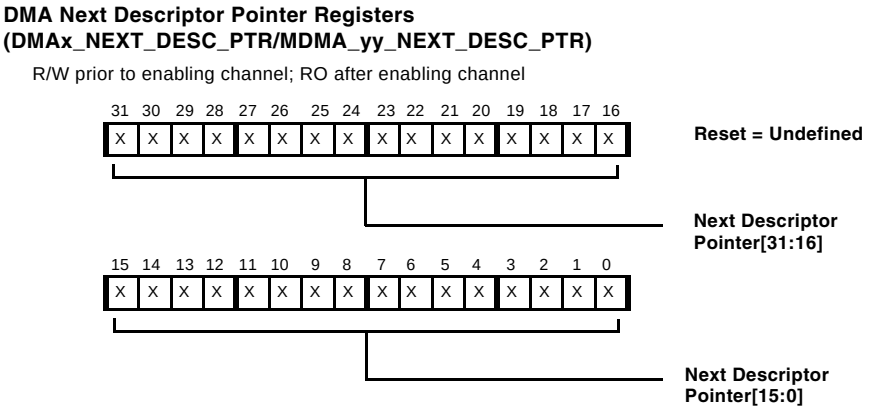

DMA Next Descriptor Pointer Registers
(DMAx_NEXT_DESC_PTR/MDMA_yy_NEXT_DESC_PTR)
R/W prior to enabling channel; RO after enabling channel
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Next Descriptor Pointer[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Next Descriptor Pointer[15:0]

Figure 6-17. DMA Current Descriptor Pointer Registers

HMDMA Registers

Some processors have two HMDMA blocks, while others have none. See the "Unique Behavior for the ADSP-BF52x Processor" on page 6-105 to determine whether this feature is applicable to your product. HMDMA0 is associated with MDMA0, and HMDMA1 is associated with MDMA1.

Handshake MDMA Control Registers (HMDMAx_CONTROL)

The HMDMAx_CONTROL register, shown in Figure 6-18, is used to set up HMDMA parameters and operating modes.

The DRQ[1:0] field is used to control the priority of the MDMA channel when the HMDMA is disabled, that is, when handshake control is not being used (see Table 6-6).

Table 6-6. DRQ[1:0] Values

DRQ[1:0]	Priority	Description
00	Disabled	The MDMA request is disabled.
01	Enabled/S	Normal MDMA channel priority. The channel in this mode is limited to single memory transfers separated by one idle system clock. Request single transfer from MDMA channel.
10	Enabled/M	Normal MDMA channel functionality and priority. Request multiple transfers from MDMA channel (default).
11	Urgent	The MDMA channel priority is elevated to urgent. In this state, it has higher priority for memory access than non-urgent channels. If two channels are both urgent, the lower-numbered channel has priority.

The RBC bit forces the BCOUNT register to be reloaded with the BCINIT value while the module is already active. Do not set this bit in the same write that sets the HMDMAEN bit to active.

Handshake MDMA Control Registers (HMDMAx_CONTROL)

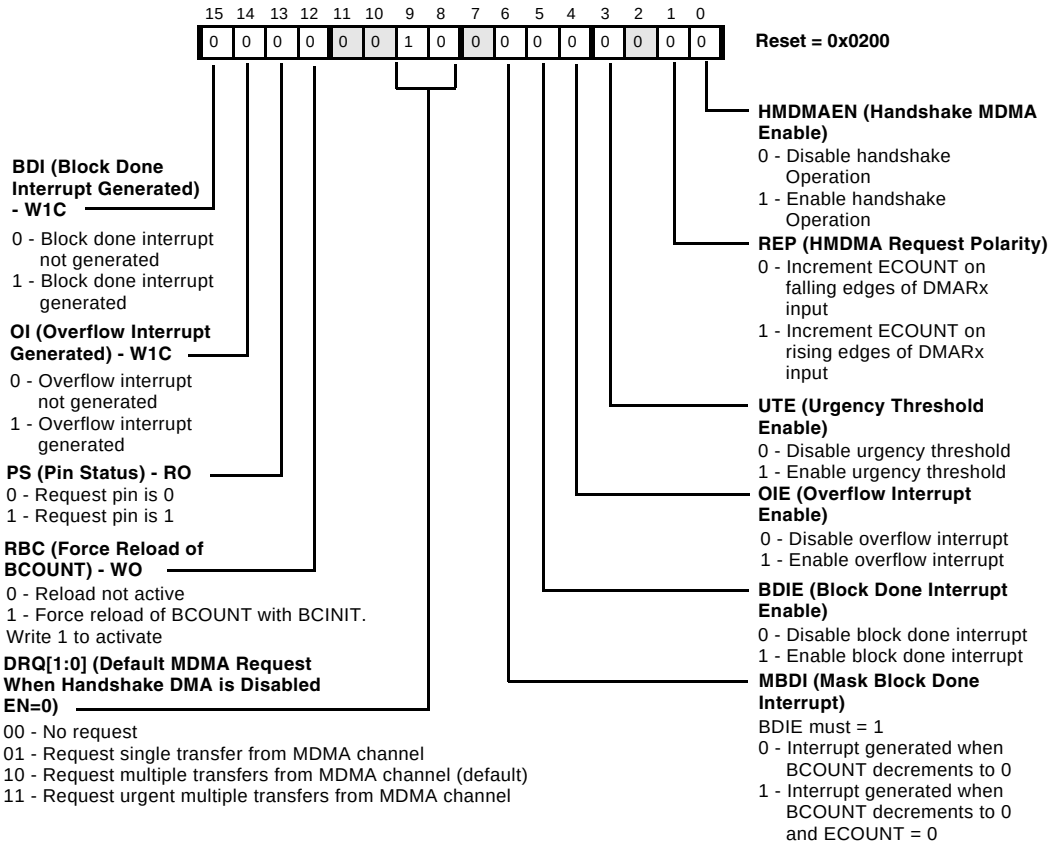

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
Reset = 0x0200
HMDMAEN (Handshake MDMA Enable)
0 - Disable handshake Operation
1 - Enable handshake Operation
REP (HMDMA Request Polarity)
0 - Increment ECOUNT on falling edges of DMARx input
1 - Increment ECOUNT on rising edges of DMARx input
UTE (Urgency Threshold Enable)
0 - Disable urgency threshold
1 - Enable urgency threshold
OIE (Overflow Interrupt Enable)
0 - Disable overflow interrupt
1 - Enable overflow interrupt
BDIE (Block Done Interrupt Enable)
0 - Disable block done interrupt
1 - Enable block done interrupt
MBDI (Mask Block Done Interrupt)
BDIE must = 1
0 - Interrupt generated when BCOUNT decrements to 0
1 - Interrupt generated when BCOUNT decrements to 0 and ECOUNT = 0
BDI (Block Done Interrupt Generated) - W1C
0 - Block done interrupt not generated
1 - Block done interrupt generated
OI (Overflow Interrupt Generated) - W1C
0 - Overflow interrupt not generated
1 - Overflow interrupt generated
PS (Pin Status) - RO
0 - Request pin is 0
1 - Request pin is 1
RBC (Force Reload of BCOUNT) - WO
0 - Reload not active
1 - Force reload of BCOUNT with BCINIT. Write 1 to activate
DRQ[1:0] (Default MDMA Request When Handshake DMA is Disabled EN=0)
00 - No request
01 - Request single transfer from MDMA channel
10 - Request multiple transfers from MDMA channel (default)
11 - Request urgent multiple transfers from MDMA channel

Figure 6-18. Handshake MDMA Control Registers

Handshake MDMA Initial Block Count Registers (HMDMAx_BCINIT)

The HMDMAx_BCINIT register, shown in Figure 6-19, holds the number of transfers to do per edge of the DMARx control signal.

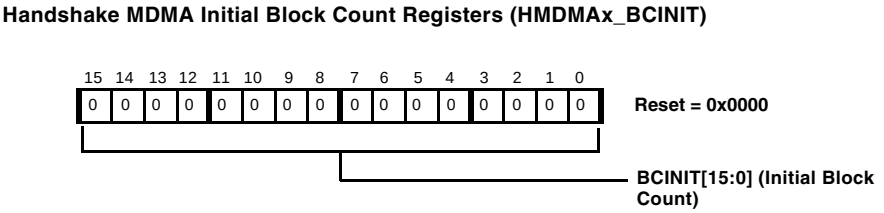

Handshake MDMA Initial Block Count Registers (HMDMAx_BCINIT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
BCINIT[15:0] (Initial Block Count)

Figure 6-19. Handshake MDMA Initial Block Count Registers

Handshake MDMA Current Block Count Registers (HMDMAx_BCOUNT)

The HMDMAx_BCOUNT register, shown in Figure 6-20, holds the number of transfers remaining for the current edge. MDMA requests are generated if this count is greater than 0.

Examples:

- 0000 = 0 transfers remaining
- FFFF = 65535 transfers remaining

The BCOUNT field is loaded with BCINIT when ECOUNT is greater than 0 and BCOUNT is expired (0). Also, if the RBC bit in the HMDMAx_CONTROL register is written to 1, BCOUNT is loaded with BCINIT. The BCOUNT field is decremented with each MDMA grant. It is cleared when HMDMA is disabled.

A block done interrupt is generated when BCOUNT decrements to 0. If the MBDI bit in the HMDMAx_CONTROL register is set, the interrupt is suppressed

until ECOUNT is 0. If BCINIT is 0, no block done interrupt is generated, since no DMA requests were generated or grants received.

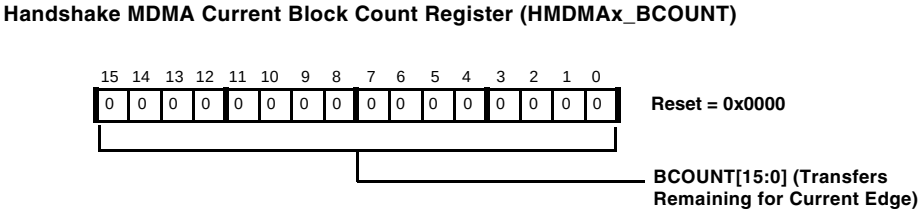

Handshake MDMA Current Block Count Register (HMDMAx_BCOUNT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
BCOUNT[15:0] (Transfers Remaining for Current Edge)

Figure 6-20. Handshake MDMA Current Block Count Registers

Handshake MDMA Current Edge Count Registers (HMDMAx_ECOUNT)

The HMDMAx_ECOUNT register, shown in Figure 6-21, holds a signed number of edges remaining to be serviced. This number is in a signed two's complement representation. When an edge is detected on the respective DMARx input, requests occur if this count is greater than or equal to 0 and BCOUNT is greater than 0.

When the handshake mode is enabled, ECOUNT is loaded and the resulting number of requests is:

Number of edges + N,

where N is the number loaded from ECINIT. The number N can be positive or negative. Examples:

- 0x7FFF = 32,767 edges remaining
- 0x0000 = 0 edges remaining
- 0x8000 = –32,768: ignore the next 32,768 edges

Each time that BCOUNT expires, ECOUNT is decremented and BCOUNT is reloaded from BCINIT. When a handshake request edge is detected, ECOUNT is incremented. The ECOUNT field is cleared when HMDMA is disabled.

Handshake MDMA Current Edge Count Register (HMDMAx_ECOUNT)

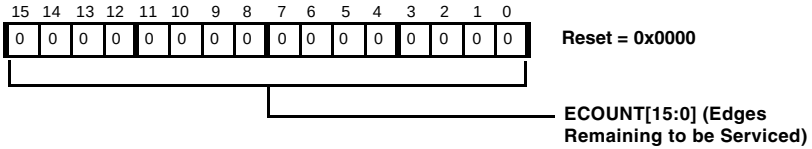


Figure 6-21. Handshake MDMA Current Edge Count Registers

Handshake MDMA Initial Edge Count Registers (HMDMAx_ECINIT)

The HMDMAx_ECINIT register, shown in Figure 6-22, holds a signed number that is loaded into HMDMAx_ECOUNT when handshake DMA is enabled. This number is in a signed two's complement representation.

Handshake MDMA Initial Edge Count Registers (HMDMAx_ECINIT)

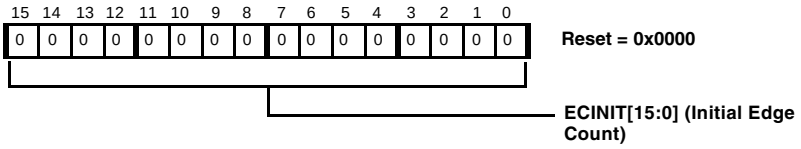


Figure 6-22. Handshake MDMA Initial Edge Count Registers

Handshake MDMA Edge Count Urgent Registers (HMDMAx_ECURGENT)

The HMDMAx_ECURGENT register, shown in Figure 6-23, holds the urgent threshold. If the ECOUNT field in the HMDMAx_ECOUNT register is greater than

this threshold, the MDMA request is urgent and might get higher priority.

Handshake MDMA Edge Count Urgent Registers (HMDMAx_ECURGENT)

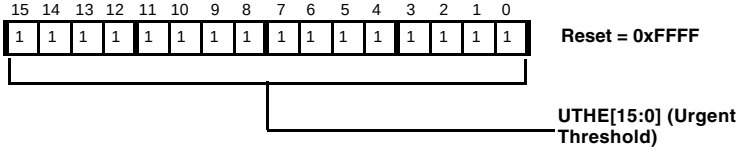


Figure 6-23. Handshake MDMA Edge Count Urgent Registers

Handshake MDMA Edge Count Overflow Interrupt Registers (HMDMAx_ECOVERFLOW)

The HMDMAx_ECOVERFLOW register, shown in Figure 6-24, holds the interrupt threshold. If the ECOUNT field in the HMDMAx_ECOUNT register is greater than this threshold, an overflow interrupt is generated.

Handshake MDMA Edge Count Overflow Interrupt Registers (HMDMAx_ECOVERFLOW)

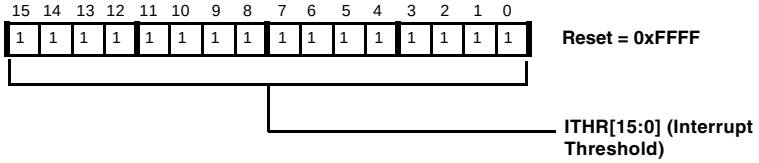


Figure 6-24. Handshake MDMA Edge Count Overflow Interrupt Registers

DMA Traffic Control Registers (DMA_TC_PER and DMA_TC_CNT)

The DMA_TC_PER register (see Figure 6-25) and the DMA_TC_CNT register (see Figure 6-26) work with other DMA registers to define traffic control.

DMA_TC_PER Register

DMA Traffic Control Counter Period Register (DMA_TC_PER)

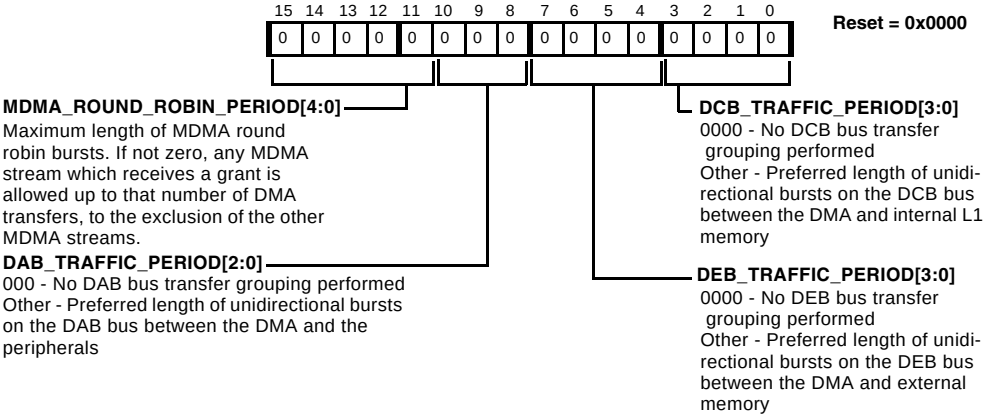

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
MDMA_ROUND_ROBIN_PERIOD[4:0]
Maximum length of MDMA round robin bursts. If not zero, any MDMA stream which receives a grant is allowed up to that number of DMA transfers, to the exclusion of the other MDMA streams.
DAB_TRAFFIC_PERIOD[2:0]
000 - No DAB bus transfer grouping performed
Other - Preferred length of unidirectional bursts on the DAB bus between the DMA and the peripherals
DCB_TRAFFIC_PERIOD[3:0]
0000 - No DCB bus transfer grouping performed
Other - Preferred length of unidirectional bursts on the DCB bus between the DMA and internal L1 memory
DEB_TRAFFIC_PERIOD[3:0]
0000 - No DEB bus transfer grouping performed
Other - Preferred length of unidirectional bursts on the DEB bus between the DMA and external memory

Figure 6-25. DMA Traffic Control Counter Period Register

DMA_TC_CNT Register

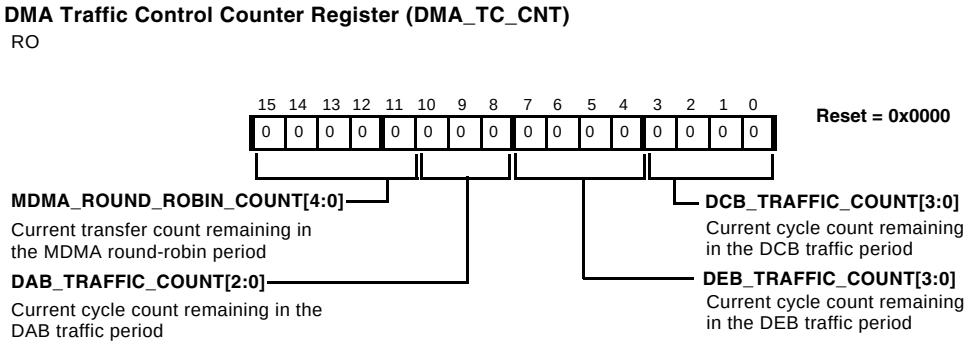

DMA Traffic Control Counter Register (DMA_TC_CNT)
RO
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
MDMA_ROUND_ROBIN_COUNT[4:0]
Current transfer count remaining in the MDMA round-robin period
DAB_TRAFFIC_COUNT[2:0]
Current cycle count remaining in the DAB traffic period
DCB_TRAFFIC_COUNT[3:0]
Current cycle count remaining in the DCB traffic period
DEB_TRAFFIC_COUNT[3:0]
Current cycle count remaining in the DEB traffic period

Figure 6-26. DMA Traffic Control Counter Register

The MDMA_ROUND_ROBIN_COUNT field shows the current transfer count remaining in the MDMA round-robin period. It initializes to MDMA_ROUND_ROBIN_PERIOD whenever DMA_TC_PER is written, whenever a different MDMA stream is granted, or whenever every MDMA stream is idle. It then counts down to 0 with each MDMA transfer. When this count decrements from 1 to 0, the next available MDMA stream is selected.

The DAB_TRAFFIC_COUNT field shows the current cycle count remaining in the DAB traffic period. It initializes to DAB_TRAFFIC_PERIOD whenever DMA_TC_PER is written, or whenever the DAB bus changes direction or becomes idle. It then counts down from DAB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DAB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DAB access is treated preferentially, which may result in a direction change. When this count is 0 and a DAB bus access occurs, the count is reloaded from DAB_TRAFFIC_PERIOD to begin a new burst.

The DEB_TRAFFIC_COUNT field shows the current cycle count remaining in the DEB traffic period. It initializes to DEB_TRAFFIC_PERIOD whenever

DMA_TC_PER is written or whenever the DEB bus changes direction or becomes idle. It then counts down from DEB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DEB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DEB access is treated preferentially, which may result in a direction change. When this count is 0 and a DEB bus access occurs, the count is reloaded from DEB_TRAFFIC_PERIOD to begin a new burst.

The DCB_TRAFFIC_COUNT field shows the current cycle count remaining in the DCB traffic period. It initializes to DCB_TRAFFIC_PERIOD whenever DMA_TC_PER is written or whenever the DCB bus changes direction or becomes idle. It then counts down from DCB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DCB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DCB access is treated preferentially, which may result in a direction change. When this count is 0 and a DCB bus access occurs, the count is reloaded from DCB_TRAFFIC_PERIOD to begin a new burst.

Programming Examples

The following examples illustrate memory DMA and handshaked memory DMA basics. Examples for peripheral DMAs can be found in the respective peripheral chapters.

Register-Based 2-D Memory DMA

Listing 6-1 shows a register-based, two-dimensional MDMA. While the source channel processes linearly, the destination channel re-sorts elements by transposing the two-dimensional data array. See Figure 6-27.

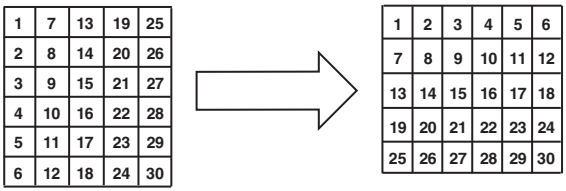


Figure 6-27. DMA Example, 2-D Array

The two arrays reside in two different L1 data memory blocks. However, the arrays could reside in any internal or external memory, including L1 instruction memory and SDRAM. For the case where the destination array resided in SDRAM, it is a good idea to let the source channel re-sort elements and to let the destination buffer store linearly.

Listing 6-1. Register-Based 2-D Memory DMA

```
#include <defBF527.h>/*For ADSP-BF527 product, as an example.*/
#define X 5
#define Y 6

.section L1_data_a;
.byte2 aSource[X*Y] =
    1,  7, 13, 19, 25,
    2,  8, 14, 20, 26,
    3,  9, 15, 21, 27,
    4, 10, 16, 22, 28,
    5, 11, 17, 23, 29,
```

```
    6, 12, 18, 24, 30;

.section L1_data_b;
.byte2 aDestination[X*Y];

.section L1_code;
.global _main;
_main:
    p0.l = lo(MDMA_S0_CONFIG);
    p0.h = hi(MDMA_S0_CONFIG);
    call memdma_setup;
    call memdma_wait;
_main.forever:
    jump _main.forever;
_main.end:
```

The setup routine shown in Listing 6-2 initializes either MDMA0 or MDMA1, depending on whether the MMR address of `MDMA_S0_CONFIG` or `MDMA_S1_CONFIG` is passed in the `P0` register. Note that the source channel is enabled before the destination channel. Also, it is common to synchronize interrupts with the destination channel because only those interrupts indicate completion of both DMA read and write operations.

Listing 6-2. Two-Dimensional Memory DMA Setup Example

```
memdma_setup:
    [--sp] = r7;
/* setup 1D source DMA for 16-bit transfers */
    r7.l = lo(aSource);
    r7.h = hi(aSource);
    [p0 + MDMA_S0_START_ADDR - MDMA_S0_CONFIG] = r7;
    r7.l = 2;
    w[p0 + MDMA_S0_X_MODIFY - MDMA_S0_CONFIG] = r7;
     r7.l = X * Y;
    w[p0 + MDMA_S0_X_COUNT - MDMA_S0_CONFIG] = r7;
```

```
    r7.l = WDSIZE_16 | DMAEN;
    w[p0] = r7;
/* setup 2D destination DMA for 16-bit transfers */
    r7.l = lo(aDestination);
    r7.h = hi(aDestination);
    [p0 + MDMA_D0_START_ADDR - MDMA_S0_CONFIG] = r7;
    r7.l = 2*Y;
    w[p0 + MDMA_D0_X_MODIFY - MDMA_S0_CONFIG] = r7;
    r7.l = Y;
    w[p0 + MDMA_D0_Y_COUNT - MDMA_S0_CONFIG] = r7;
    r7.l = X;
    w[p0 + MDMA_D0_X_COUNT - MDMA_S0_CONFIG] = r7;
    r7.l = -2 * (Y * (X-1) - 1);
    w[p0 + MDMA_D0_Y_MODIFY - MDMA_S0_CONFIG] = r7;
    r7.l = DMA2D | DI_EN | WDSIZE_16 | WNR | DMAEN;
    w[p0 + MDMA_D0_CONFIG - MDMA_S0_CONFIG] = r7;
    r7 = [sp++];
    rts;
memdma_setup.end:
```

For simplicity the example shown in Listing 6-3 polls the DMA status rather than using interrupts, which is the normal case in a real application.

Listing 6-3. Polling DMA Status

```
memdma_wait:
    [--sp] = r7;
memdma_wait.test:
    r7 = w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] (z);
    CC = bittst (r7, bitpos(DMA_DONE));
    if !CC jump memdma_wait.test;
    r7 = DMA_DONE (z);
    w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] = r7;
    r7 = [sp++];
```

```
    rts;
memdma_wait.end:
```

Initializing Descriptors in Memory

Descriptor-based DMAs expect the descriptor data to be available in memory by the time the DMA is enabled. Often, the descriptors are programmed by software at run-time. Many times, however, the descriptors—or at least large portions of them—can be static and therefore initialized at boot time. How to set up descriptors in global memory depends heavily on the programming language and the tool set used. The following examples show how this is best performed in the VisualDSP++ tools' assembly language.

Listing 6-4 uses multiple variables of either 16-bit or 32-bit size to describe DMA descriptors. This example has two descriptors in small list flow mode that point to each other. At the end of the second work unit, an interrupt is generated without discontinuing the DMA processing. The trailing `.end` label is required to let the linker know that a descriptor forms a logical unit. It prevents the linker from removing variables when optimizing.

Listing 6-4. Two Descriptors in Small List Flow Mode

```
.section sdram;
.byte2 arrBlock1[0x400];
.byte2 arrBlock2[0x800];

.section L1_data_a;
.byte2 descBlock1 = lo(descBlock2);
.var   descBlock1.addr = arrBlock1;
.byte2 descBlock1.cfg = FLOW_SMALL|NDSIZE_5|WDSIZE_16|DMAEN;
.byte2 descBlock1.len = length(arrBlock1);
```

```
        descBlock1.end:

.byte2 descBlock2 = lo(descBlock1);
.var   descBlock2.addr = arrBlock2;
.byte2 descBlock2.cfg =
FLOW_SMALL|NDSIZE_5|DI_EN|WDSIZE_16|DMAEN;
.byte2 descBlock2.len = length(arrBlock2);
        descBlock2.end:
```

Another method featured by the VisualDSP++ tools takes advantage of C-style structures in global header files. The header file `descriptors.h` could look like Listing 6-5.

Listing 6-5. Header File to Define Descriptor Structures

```
#ifndef __INCLUDE_DESCRIPTORS__
#define __INCLUDE_DESCRIPTORS__
#ifdef _LANGUAGE_C
typedef struct {
    void  *pStart;
    short dConfig;
    short dXCount;
    short dXModify;
    short dYCount;
    short dYModify;
} dma_desc_arr;

typedef struct {
    void  *pNext;
    void  *pStart;
    short dConfig;
    short dXCount;
    short dXModify;
    short dYCount;
    short dYModify;
```

```
} dma_desc_list;

#endif // _LANGUAGE_C
#endif // __INCLUDE_DESCRIPTORS__
```

Note that near pointers are not natively supported by the C language and, thus, pointers are always 32 bits wide. Therefore, the scheme above cannot be used directly for small list mode without giving up pointer syntax. The variable definition file is required to import the C-style header file and can finally take advantage of the structures. See Listing 6-6.

Listing 6-6. Using Descriptor Structures

```
#include "descriptors.h"
.import "descriptors.h";

.section L1_data_a;
.align 4;
.var arrBlock3[N];
.var arrBlock4[N];

.struct dma_desc_list descBlock3 = {
    descBlock4, arrBlock3,
    FLOW_LARGE | NDSIZE_7 | WDSIZE_32 | DMAEN,
    length(arrBlock3), 4,
    0, 0        /* unused values */
};

.struct dma_desc_list descBlock4 = {
    descBlock3, arrBlock4,
    FLOW_LARGE | NDSIZE_7 | DI_EN | WDSIZE_32 | DMAEN,
    length(arrBlock4), 4,
    0, 0        /* unused values */
};
```

Software-Triggered Descriptor Fetch Example

Listing 6-7 demonstrates a large list of descriptors that provide FLOW = 0 (stop mode) configuration. Consequently, the DMA stops by itself as soon as the work unit has finished. Software triggers the next work unit by simply writing the proper value into the DMA configuration registers. Since these values instruct the DMA controller to fetch descriptors in large list mode, the DMA immediately fetches the descriptor, thus overwriting the configuration value again with the new settings when it is started.

Note the requirement that source and destination channels stop after the same number of transfers. Between stops, the two channels can have completely individual structures.

Listing 6-7. Software-Triggered Descriptor Fetch

```
.import "descriptors.h";

#define N 4
.section L1_data_a;
.byte2 arrSource1[N] = { 0x1001, 0x1002, 0x1003, 0x1004 };
.byte2 arrSource2[N] = { 0x2001, 0x2002, 0x2003, 0x2004 };
.byte2 arrSource3[N] = { 0x3001, 0x3002, 0x3003, 0x3004 };
.byte2 arrDest1[N];
.byte2 arrDest2[2*N];

.struct dma_desc_list descSource1 = {
    descSource2, arrSource1,
    WDSIZE_16 | DMAEN,
    length(arrSource1), 2,
    0, 0        /* unused values */
};
.struct dma_desc_list descSource2 = {
    descSource3, arrSource2,
    FLOW_LARGE | NDSIZE_7 | WDSIZE_16 | DMAEN,
```

```
    length(arrSource2), 2,
    0, 0        /* unused values */
};
.struct dma_desc_list descSource3 = {
    descSource1, arrSource3,
    WDSIZE_16 | DMAEN,
    length(arrSource3), 2,
    0, 0        /* unused values */
};
.struct dma_desc_list descDest1 = {
    descDest2, arrDest1,
    DI_EN | WDSIZE_16 | WNR | DMAEN,
    length(arrDest1), 2,
    0, 0        /* unused values */
};
.struct dma_desc_list descDest2 = {
    descDest1, arrDest2,
    DI_EN | WDSIZE_16 | WNR | DMAEN,
    length(arrDest2), 2,
    0, 0        /* unused values */
};

.section L1_code;
_main:
/* write descriptor address to next descriptor pointer */
    p0.h = hi(MDMA_S0_CONFIG);
    p0.l = lo(MDMA_S0_CONFIG);
    r0.h = hi(descDest1);
    r0.l = lo(descDest1);
    [p0 + MDMA_D0_NEXT_DESC_PTR - MDMA_S0_CONFIG] = r0;
    r0.h = hi(descSource1);
    r0.l = lo(descSource1);
    [p0 + MDMA_S0_NEXT_DESC_PTR - MDMA_S0_CONFIG] = r0;
```

```
/* start first work unit */
    r6.l = FLOW_LARGE|NDSIZE_7|WDSIZE_16|DMAEN;
    w[p0 + MDMA_S0_CONFIG - MDMA_S0_CONFIG] = r6;
    r7.l = FLOW_LARGE|NDSIZE_7|WDSIZE_16|WNR|DMAEN;
    w[p0 + MDMA_D0_CONFIG - MDMA_S0_CONFIG] = r7;

/* wait until destination channel has finished and W1C latch */
_main.wait:
    r0 = w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] (z);
    CC = bittst (r0, bitpos(DMA_DONE));
    if !CC jump _main.wait;
    r0.l = DMA_DONE;
    w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] = r0;

/* wait for any software or hardware event here */

/* start next work unit */
    w[p0 + MDMA_S0_CONFIG - MDMA_S0_CONFIG] = r6;
    w[p0 + MDMA_D0_CONFIG - MDMA_S0_CONFIG] = r7;
    jump _main.wait;
_main.end:
```

Handshaked Memory DMA Example

The functional block for the handshaked MDMA operation can be considered completely separately from the MDMA channels themselves. Therefore the following HMDMA setup routine can be combined with any of the MDMA examples discussed above. Be sure that the HMDMA module is enabled before the MDMA channels.

Listing 6-8 enables the HMDMA1 block, which is controlled by the `DMAR1` pin and is associated with the MDMA1 channel pair.

Listing 6-8. HMDMA1 Block Enable

```
/* optionally, enable all four bank select strobes */
    p1.l = lo(EBIU_AMGCTL);
    p1.h = hi(EBIU_AMGCTL);
    r0.l = 0x0009;
    w[p1] = r0;

/* function enable for DMAR1 */
    p1.l = lo(PORTG_FER);
    r0.l = PG12;
    w[p1] = r0;
    p1.l = lo(PORTG_MUX);
    r0.l = 0x0000;
    w[p1] = r0;

/* every single transfer requires one DMAR1 event */
    p1.l = lo(HMDMA1_BCINIT);
    r0.l = 1;
    w[p1] = r0;

/* start with balanced request counter */
    p1.l = lo(HMDMA1_ECINIT);
    r0.l = 0;
    w[p1] = r0;

/* enable for rising edges */
    p1.l = lo(HMDMA1_CONTROL);
    r2.l = REP | HMDMAEN;
    w[p1] = r2;
```

If the HMDMA is intended to copy from internal memory to external devices, the above setup is sufficient. If, however, the data flow is from outside the processor to internal memory, then this small issue must be considered—the HMDMA only controls the destination channel of the

memory DMA. It does not gate requests to the source channel at all. Thus, as soon as the source channel is enabled, it starts filling the DMA FIFO immediately. In 16-bit DMA mode, this results in eight read strobes on the EBIU even before the first DMAR1 event has been detected. In other words, the transferred data and the DMAR1 strobes are eight positions off. The example in Listing 6-9 delays processing until eight DMAR1 requests have been received. By doing so, the transmitter is required to add eight trailing dummy writes after all data words have been sent. This is because the transmit channel still has to drain the DMA FIFO.

Listing 6-9. HMDMA With Delayed Processing

```
/* wait for eight requests */
    p1.l = lo(HMDMA1_ECOUNT);
    r0 = 7 (z);
initial_requests:
    r1 = w[p1] (z);
    CC = r1 < r0;
    if CC jump initial_requests;

/* disable and reenable to clear edge count */
    p1.l = lo(HMDMA1_CONTROL);
    r0.l = 0;
    w[p1] = r0;
    w[p1] = r2;
```

If the polling operation shown in Listing 6-9 is too expensive, an interrupt version of it can be implemented by using the HMDMA overflow feature. Temporarily set the HMDMAx_OVERFLOW register to eight.

Unique Behavior for the ADSP-BF52x Processor

Figure 6-28 provides a block diagram of the DMA controller.

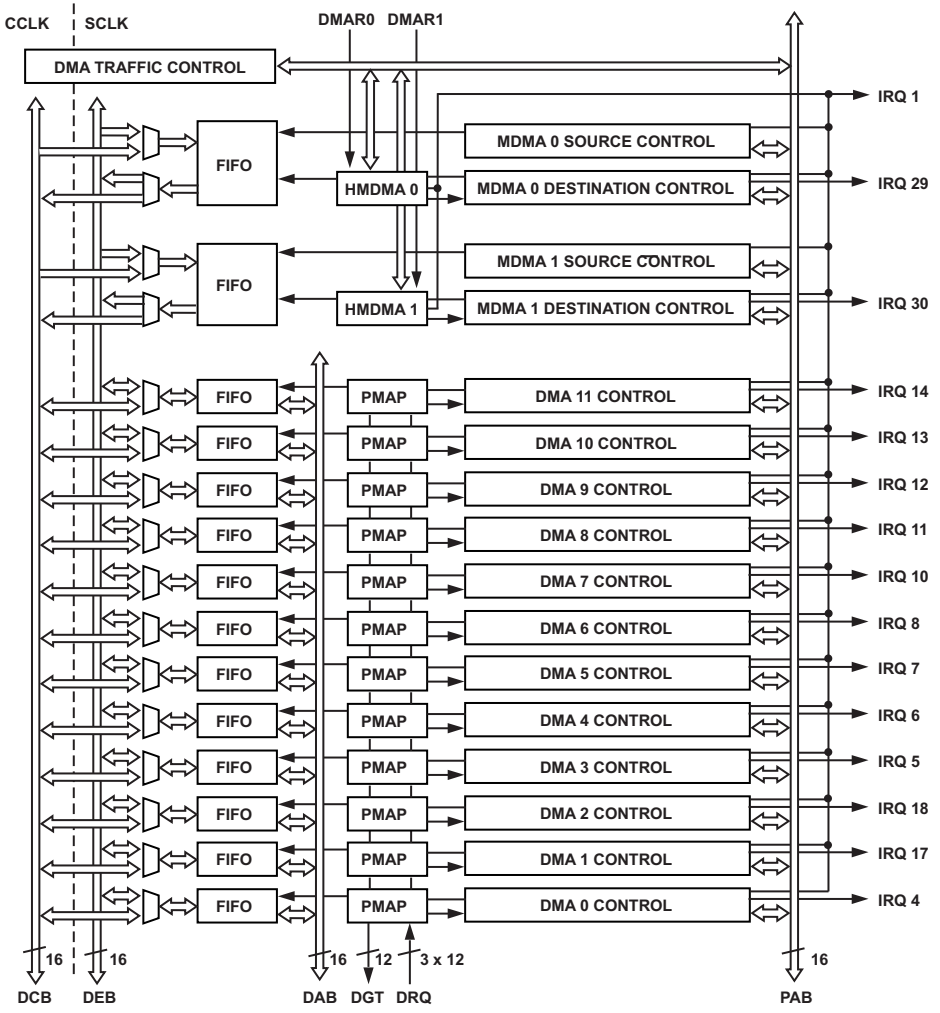


Figure 6-28. ADSP-BF52x DMA Controller Block Diagram

Static Channel Prioritization

The default configuration shown in Table 6-7 can be changed by altering the 4-bit PMAP field in the DMAx_PERIPHERAL_MAP registers for the peripheral DMA channels.

Table 6-7. Priority and Default Mapping of Peripheral to DMA

Priority	DMA Channel	PMAP Default Value	Peripheral Mapped by Default
Highest	DMA 0	0x0	PPI receive/transmit or NFC
	DMA 1	0x1	Ethernet MAC receive[1] or HOSTDP
	DMA 2	0x2	Ethernet MAC transmit[1] or NFC
	DMA 3	0x3	SPORT0 receive[1]
	DMA 4	0x4	SPORT0 transmit[1]
	DMA 5	0x5	SPORT1 receive[1]
	DMA 6	0x6	SPORT1 transmit[1]
	DMA 7	0x7	SPI
	DMA 8	0x8	UART0 receive
	DMA 9	0x9	UART0 transmit
	DMA 10	0xA	UART1 receive
	DMA 11	0xB	UART1 transmit
	MDMA D0[1]	N/A	N/A
	MDMA S0[1]	N/A	N/A
	MDMA D1[1]	N/A	N/A
Lowest	MDMA S1[1]	N/A	N/A

1 Can be set to use 32-bit DMA. The 32-bit DMA setting causes the DMA channel to do back-to-back 16-bit transactions which can lead to improved performance.

Although only the ADSP-BF526 and ADSP-BF527 processors feature the Ethernet MAC module, the DMA1 and DMA2 channels are still present on all parts and can be used for the HOSTDP or NFC. Peripherals which share a set of multiplexed pins can also share a DMA channel. Whichever peripheral is configured in `PORTx_MUX` to utilize the shared pins and is enabled will get the shared DMA channel. It is up to the user to ensure that multiple peripherals which share pins aren't enabled simultaneously.

DMA Control Commands

The ADSP-BF52x processors have two DMA-management-capable peripherals; the Ethernet MAC and the Host DMA Port. Refer to the "Ethernet MAC", and "Host DMA Port" chapters for a description of how these peripherals use DMA control commands.

Handshaked Memory DMA Operation

HMDMA Interrupts

All interrupt sources are routed to the global DMA error interrupt channel.

7 EXTERNAL BUS INTERFACE UNIT

The external bus interface unit (EBIU) provides glueless interfaces to external memories. The processor supports Synchronous DRAM (SDRAM) including mobile SDRAM. The EBIU also supports asynchronous interfaces such as SRAM, ROM, FIFOs, flash memory, and ASIC/FPGA designs.

EBIU Overview

The EBIU services requests for external memory from the core or from a DMA channel. The priority of the requests is determined by the external bus controller. The address of the request determines whether the request is serviced by the EBIU SDRAM controller or the EBIU asynchronous memory controller.

The DMA controller provides high-bandwidth data movement capability. The Memory DMA (MDMA) channels can perform block transfers of code or data between the internal memory and the external memory spaces. The MDMA channels also feature a Handshake Operation mode (HMDMA) via dual external DMA request pins. When used in conjunction with the EBIU, this functionality can be used to interface high-speed external devices, such as FIFOs and USB 2.0 controllers, in an automatic manner. For more information on HMDMA and the external DMA request pins, please refer to Chapter 6, “Direct Memory Access”.

The EBIU is clocked by the system clock (SCLK). All synchronous memories interfaced to the processor operate at the SCLK frequency. The ratio between core clock frequency (CCLK) and SCLK frequency is programmable

using a Phase Locked Loop (PLL) system Memory-Mapped Register (MMR). For more information, see “Core Clock/System Clock Ratio Control” on page 18-5.

The external memory space is shown in Figure 7-1. One memory region is dedicated to SDRAM support. SDRAM interface timing and the size of the SDRAM region are programmable. The SDRAM memory space can range in size from 16M byte to 128M byte.

The start address of the SDRAM memory space is 0x0000 0000. The area from the end of the SDRAM memory space up to address 0x2000 0000 is reserved.

The next four regions are dedicated to supporting asynchronous memories. Each asynchronous memory region can be independently programmed to support different memory device characteristics. Each region has its own memory select output pin from the EBIU.

The next region is reserved memory space. References to this region do not generate external bus transactions. Writes have no effect on external memory values, and reads return undefined values. The EBIU generates an error response on the internal bus, which will generate a hardware

exception for a core access or will optionally generate an interrupt from a DMA channel.

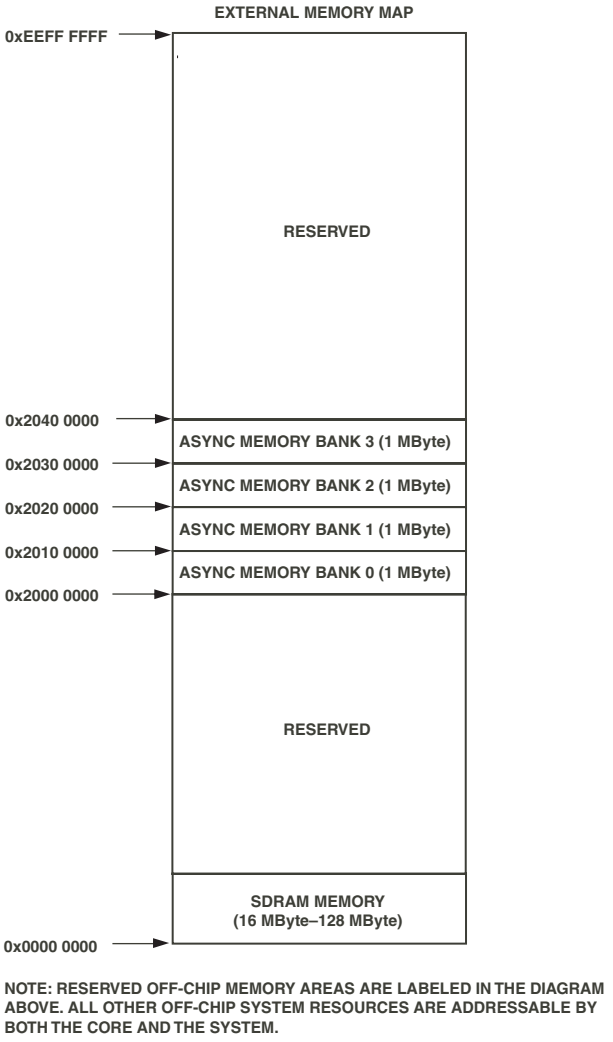

EXTERNAL MEMORY MAP
0xEEFF FFFF
RESERVED
0x2040 0000
ASYNC MEMORY BANK 3 (1 MByte)
0x2030 0000
ASYNC MEMORY BANK 2 (1 MByte)
0x2020 0000
ASYNC MEMORY BANK 1 (1 MByte)
0x2010 0000
ASYNC MEMORY BANK 0 (1 MByte)
0x2000 0000
RESERVED
SDRAM MEMORY
(16 MByte–128 MByte)
0x0000 0000
NOTE: RESERVED OFF-CHIP MEMORY AREAS ARE LABELED IN THE DIAGRAM ABOVE. ALL OTHER OFF-CHIP SYSTEM RESOURCES ARE ADDRESSABLE BY BOTH THE CORE AND THE SYSTEM.

Figure 7-1. External Memory Map

Block Diagram

Figure 7-2 is a conceptual block diagram of the EBIU and its interfaces. Signal names shown with an overbar are active low signals.

Since only one external memory device can be accessed at a time, control, address, and data pins for each memory type are multiplexed together at the pins of the device. The Asynchronous Memory Controller (AMC) and the SDRAM Controller (SDC) effectively arbitrate for the shared pin resources.

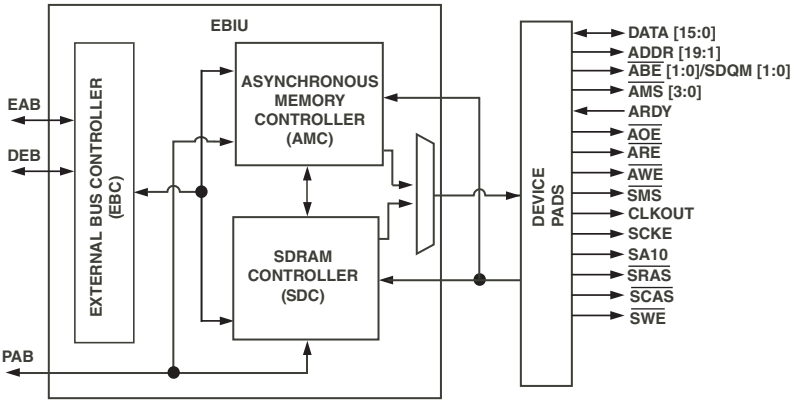


Figure 7-2. External Bus Interface Unit (EBIU)

Internal Memory Interfaces

The EBIU functions as a slave on three buses internal to the processor:

- External Access Bus (EAB), mastered by the core memory management unit on behalf of external bus requests from the core
- DMA External Bus (DEB), mastered by the DMA controller on behalf of external bus requests from any DMA channel
- Peripheral Access Bus (PAB), mastered by the core on behalf of system MMR requests from the core

These are synchronous interfaces, clocked by SCLK, as is the EBIU. The EAB provides access to both asynchronous external memory and synchronous DRAM external memory. The external access is controlled by either the AMC or the SDC, depending on the internal address used to access the EBIU. Since the AMC and SDC share the same interface to the external pins, access is sequential and must be arbitrated based on requests from the EAB.

The third bus (PAB) is used only to access the memory-mapped control and status registers of the EBIU. The PAB connects separately to the AMC and SDC. It does not need to arbitrate with, nor take access cycles from, the EAB bus.

The External Bus Controller (EBC) logic must arbitrate access requests for external memory coming from the EAB and DEB buses. The EBC logic routes read and write requests to the appropriate memory controller based on the bus selects. The AMC and SDC compete for access to the shared resources. This competition is resolved in a pipelined fashion, in the order dictated by the EBC arbiter. Transactions from the core have priority over DMA accesses in most circumstances. However, if the DMA controller detects an excessive backup of transactions, it can request its priority to be temporarily raised above the core.

Registers

There are six control registers and one status register in the EBIU. They are:

- Asynchronous memory global control register (EBIU_AMGCTL)
- Asynchronous memory bank control 0 register (EBIU_AMBCTL0)
- Asynchronous memory bank control 1 register (EBIU_AMBCTL1)
- SDRAM memory global control register (EBIU_SDGCTL)
- SDRAM memory bank control register (EBIU_SDBCTL)
- SDRAM refresh rate control register (EBIU_SDRRC)
- SDRAM control status register (EBIU_SDSTAT)

Each of these registers is described in detail in the AMC and SDC sections later in this chapter.

Shared Pins

Both the AMC and the SDC share the external interface address and data pins, as well as some of the control signals. These pins are shared:

- ADDR[19:1], address bus
- DATA[15:0], data bus
- $\overline{\text{ABE[1:0]}}$/SDQM[1:0], AMC byte enables/SDC data masks
- CLKOUT, system clock for SDC and AMC

No other signals are multiplexed between the two controllers.

System Clock

The CLKOUT pin is shared by both the SDC and AMC. Two different registers are used to control this:

- EBIU_SDGCTL register, SCTLE bit for SDC clock
- EBIU_AMGCTL register, AMCKEN bit for AMC clock

If enabling or disabling the system clock, software control for both registers is required.

Error Detection

The EBIU responds to any bus operation which addresses the range of 0x0000 0000 – 0xEEFF FFFF, even if that bus operation addresses reserved or disabled memory or functions. It responds by completing the bus operation (asserting the appropriate number of acknowledges as specified by the bus master) and by asserting the bus error signal for these error conditions:

- Any access to a disabled external memory bank
- Any access to reserved SDRAM memory space
- Any access to unpopulated SDRAM space

If the core requested the faulting bus operation, the bus error response from the EBIU is gated into the hardware error interrupt (IVHW) internal to the core (this interrupt can be masked off in the core). If a DMA master requested the faulting bus operation, then the bus error is captured in that controller and can optionally generate an interrupt to the core.

AMC Overview and Features

The following sections describe the features of the AMC.

Features

The EBIU AMC features include:

- 16-bit I/O width
- 1.8, 2.5 or 3.3 V I/O supply
- Supports up to 4M bytes of SRAM in four external banks
- AMC supports 8-bit data masking writes
- AMC has control of the EBIU while auto-refresh is performed to SDRAM
- AMC supports asynchronous access extension (ARDY pin)
- Supports instruction fetch
- Allows booting from bank 0 ($\overline{\text{AMS0}}$)

Asynchronous Memory Interface

The asynchronous memory interface allows a glueless interface to a variety of memory and peripheral types. These include SRAM, ROM, EPROM, flash memory, and FPGA/ASIC designs. Four asynchronous memory regions are supported. Each has a unique memory pin select associated with it, shown in Table 7-1.

Table 7-1. Asynchronous Memory Bank Address Range

Memory Bank Select	Address Start	Address End
$\overline{\text{AMS[3]}}$	0x2030 0000	0x203F FFFF
$\overline{\text{AMS[2]}}$	0x2020 0000	0x202F FFFF
$\overline{\text{AMS[1]}}$	0x2010 0000	0x201F FFFF
$\overline{\text{AMS[0]}}$	0x2000 0000	0x200F FFFF

Asynchronous Memory Address Decode

The address range allocated to each asynchronous memory bank is fixed at 1M bytes; however, not all of an enabled memory bank need be populated.

Accesses to unpopulated memory or partially populated AMC banks do not result in a bus error and will alias to valid AMC addresses.

The asynchronous memory signals are defined in Table 7-2. The timing of these pins is programmable to allow a flexible interface to devices of different speeds. For example interfaces, see Chapter 19, "System Design".

AMC Pin Description

The following table describes the signals associated with each interface.

Table 7-2. Asynchronous Memory Interface Signals

Pad	Pin Type [1]	Description
DATA[15:0]	I/O	External data bus
CLKOUT	O	Switches at system clock frequency. Connect to the peripheral if required.
ADDR[19:1]	O	External address bus
$\overline{\text{AMS[3:0]}}$	O	Asynchronous memory bank selects
$\overline{\text{AWE}}$	O	Asynchronous memory write enable
$\overline{\text{ARE}}$	O	Asynchronous memory read enable
$\overline{\text{AOE}}$	O	Asynchronous memory output enable In most cases, the $\overline{\text{AOE}}$ pin should be connected to the $\overline{\text{OE}}$ pin of an external memory-mapped asynchronous device. Please refer to the product data sheet for specific timing information between the $\overline{\text{AOE}}$ and $\overline{\text{ARE}}$ signals to determine which interface signal should be used in your system.

Table 7-2. Asynchronous Memory Interface Signals (Continued)

Pad	Pin Type [1]	Description
ARDY	I	Asynchronous memory ready response
$\overline{\text{ABE[1:0]}}$/SDQM[1:0]	O	Byte enables

1 Pin Types: I = Input, O = Output

AMC Description of Operation

The following sections describe the operation of the AMC.

Avoiding Bus Contention

Because the three-stated data bus is shared by multiple devices in a system, be careful to avoid contention. Contention causes excessive power dissipation and can lead to device failure. Contention occurs during the time one device is getting off the bus and another is getting on. If the first device is slow to three-state and the second device is quick to drive, the devices contend.

There are two cases where contention can occur. The first case is a read followed by a write to the same memory space. In this case, the data bus drivers can potentially contend with those of the memory device addressed by the read. The second case is back-to-back reads from two different memory spaces. In this case, the two memory devices addressed by the two reads could potentially contend at the transition between the two read operations.

To avoid contention, program the turnaround time (bank transition time) appropriately in the asynchronous memory bank control registers. This feature allows software to set the number of clock cycles between these types of accesses on a bank-by-bank basis. Minimally, the EBIU provides one cycle for the transition to occur.

External Access Extension

Each bank can be programmed to sample the ARDY input after the read or write access timer has counted down or to ignore this input signal. If enabled and disabled at the sample window, ARDY can be used to extend the access time as required.

The polarity of ARDY is programmable on a per-bank basis. Since ARDY is not sampled until an access is in progress to a bank in which the ARDY enable is asserted, ARDY does not need to be driven by default. For more information, see "Adding External Access Extension" on page 7-15.

AMC Functional Description

The following sections provide a functional description of the AMC.

Programmable Timing Characteristics

This section describes the programmable timing characteristics for the EBIU. Timing relationships depend on the programming of the AMC, no matter whether the transaction is initiated from the core or from memory DMA, or what the sequence of transactions is (read followed by read, read followed by write, and so on).

Asynchronous Reads

Figure 7-3 shows an asynchronous read bus cycle with timing programmed as setup = 2 cycles, read access = 2 cycles, hold = 1 cycle, and transition time = 1 cycle.

Asynchronous read bus cycles proceed as follows.

1. At the start of the setup period, $\overline{\text{AMS[x]}}$ and $\overline{\text{AOE}}$ assert. The address bus becomes valid. The $\overline{\text{ABE[1:0]}}$ signals are low during the read.
2. At the beginning of the read access period and after the 2 setup cycles, $\overline{\text{ARE}}$ asserts.
3. At the beginning of the hold period, read data is sampled on the rising edge of the EBIU clock. The $\overline{\text{ARE}}$ pin deasserts after this rising edge.

4. At the end of the hold period, $\overline{\texttt{AOE}}$ deasserts unless this bus cycle is followed by another asynchronous read to the same memory space. Also, $\overline{\texttt{AMS[x]}}$ deasserts unless the next cycle is to the same memory bank.

5. Unless another read of the same memory bank is queued internally, the AMC appends the programmed number of memory transition time cycles.

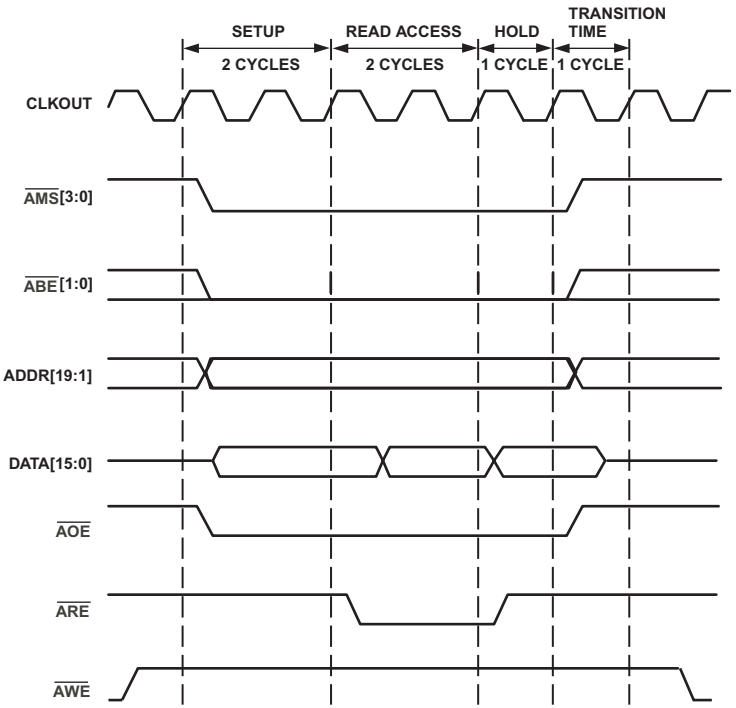


Figure 7-3. Asynchronous Read Bus Cycles

Asynchronous Writes

Figure 7-4 shows an asynchronous write bus cycle followed by an asynchronous read cycle to the same bank, with timing programmed as setup =

2 cycles, write access = 2 cycles, read access = 3 cycles, hold = 1 cycle, and transition time = 1 cycle.

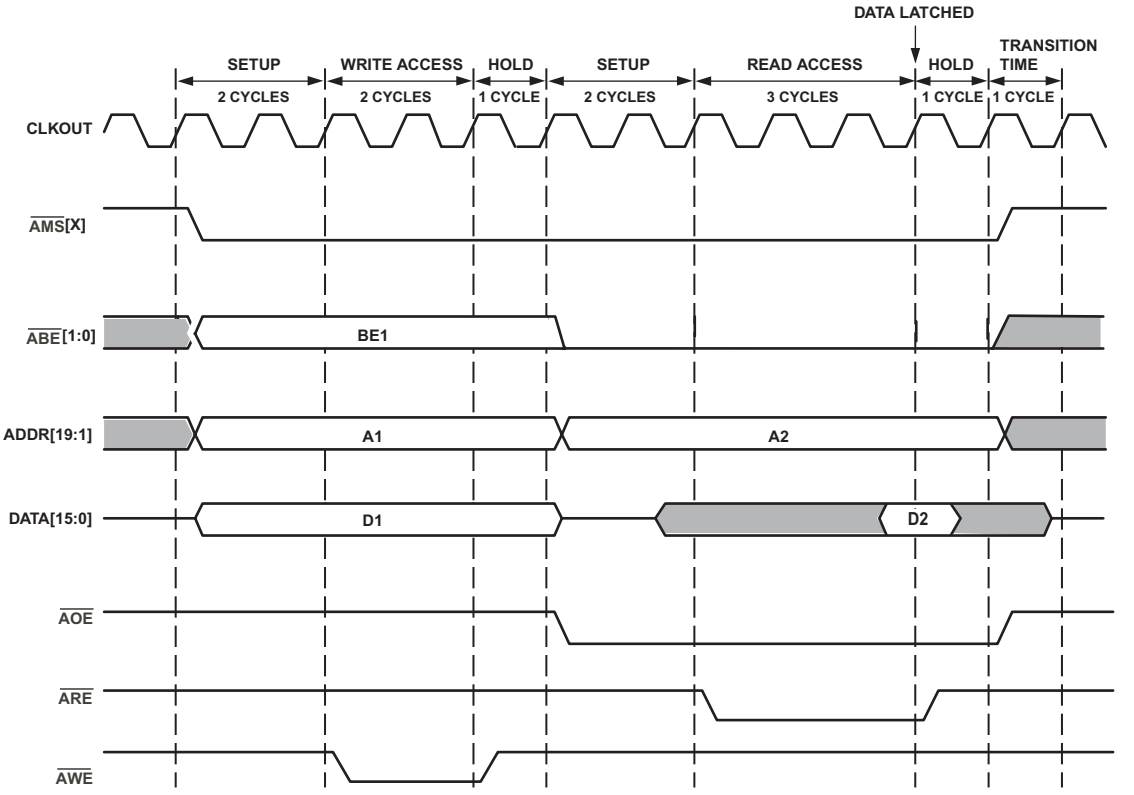


Figure 7-4. Asynchronous Write and Read Bus Cycles

Asynchronous write bus cycles proceed as follows.

1. At the start of the setup period, $\overline{\text{AMS[x]}}$, the address bus, data buses, and $\overline{\text{ABE[1:0]}}$ become valid. See "Byte Enables" on page 7-17 for more information.
2. At the beginning of the write access period, $\overline{\text{AWE}}$ asserts.
3. At the beginning of the hold period, $\overline{\text{AWE}}$ deasserts.

Asynchronous read bus cycles proceed as follows.

1. At the start of the setup period, $\overline{\text{AMS[x]}}$ and $\overline{\text{AOE}}$ assert. The address bus becomes valid. The $\overline{\text{ABE[1:0]}}$ signals are low during the read.
2. At the beginning of the read access period, $\overline{\text{ARE}}$ asserts.
3. At the beginning of the hold period, read data is sampled on the rising edge of the EBIU clock. The $\overline{\text{ARE}}$ signal deasserts after this rising edge.
4. At the end of the hold period, $\overline{\text{AOE}}$ deasserts unless this bus cycle is followed by another asynchronous read to the same memory space. Also, $\overline{\text{AMS[x]}}$ deasserts unless the next cycle is to the same memory bank.
5. Unless another read of the same memory bank is queued internally, the AMC appends the programmed number of memory transition time cycles.

Adding External Access Extension

The ARDY pin is used to insert extra wait states. The EBIU starts sampling ARDY on the clock cycle before the end of the programmed strobe period. If ARDY is sampled as deasserted, the access period is extended. The ARDY pin is then sampled on each subsequent clock edge. Read data is latched on the clock edge after ARDY is sampled as asserted. The read- or write-enable remains asserted for one clock cycle after ARDY is sampled as

asserted. An example of this behavior is shown in Figure 7-5, where setup = 2 cycles, read access = 4 cycles, and hold = 1 cycle.

The read access period must be programmed to a minimum of two cycles to make use of the ARDY input.

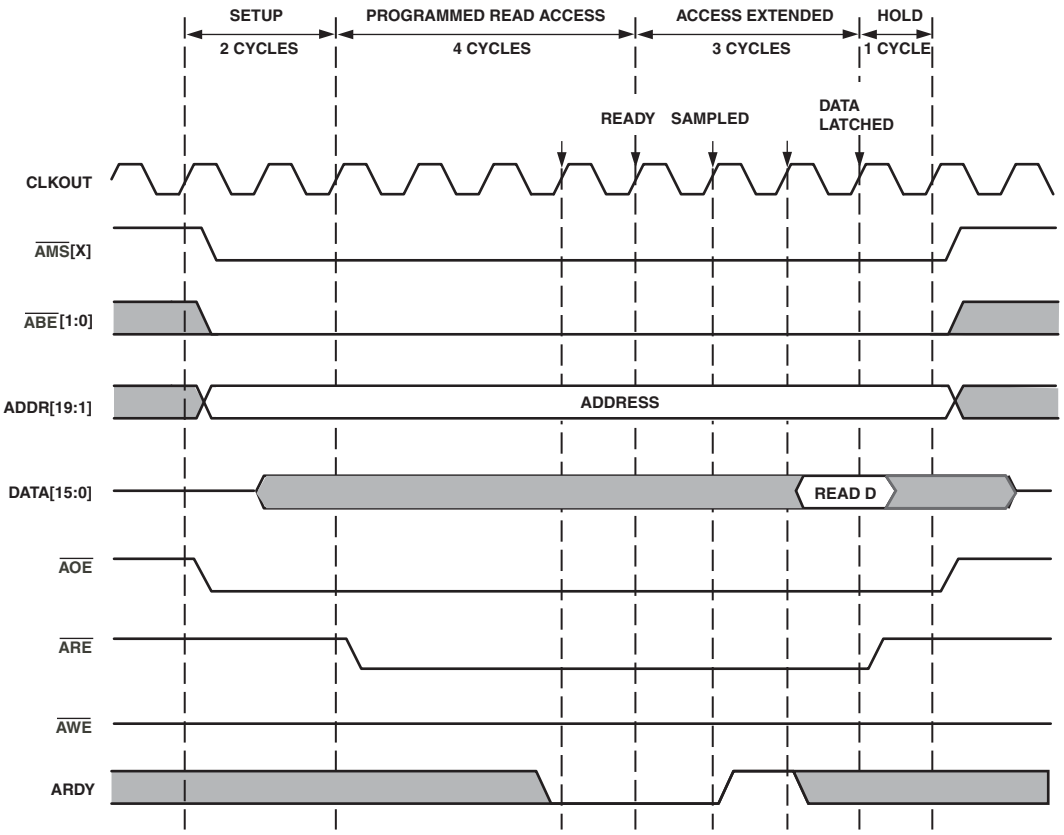


Figure 7-5. Inserting Wait States Using ARDY

Byte Enables

The AMC provides byte enable pins to allow the processor to perform efficient byte-wide arithmetic and byte-wide processing in external memory.

In general, there are two different ways to modify a single byte within the 16-bit interface. First, it can be done by a read/modify/write sequence. However, this is not very efficient because multiple accesses are required (that is, it takes many cycles for reads and writes to external memory). Another option is available where just a specific byte can be modified for a 16-bit devices using the $\overline{\texttt{ABE[1:0]}}$ pins. See Table 7-3.

The $\overline{\texttt{ABE[1:0]}}$ pins are both low during all asynchronous reads and 16-bit asynchronous writes. When an asynchronous write is made to the upper byte of a 16-bit memory, $\overline{\texttt{ABE1}}$ = 0 and $\overline{\texttt{ABE0}}$ = 1. When an asynchronous write is made to the lower byte of a 16-bit memory, $\overline{\texttt{ABE1}}$ = 1 and $\overline{\texttt{ABE0}}$ = 0.

Table 7-3. Byte Enables 8-Bit Write Accesses

Internal Address IA[0]	Internal Transfer Size	
	byte	2 bytes
0	$\overline{\texttt{ABE[1]}}$ = 1 $\overline{\texttt{ABE[0]}}$ = 0	$\overline{\texttt{ABE[1]}}$ = 0 $\overline{\texttt{ABE[0]}}$ = 0
1	$\overline{\texttt{ABE[1]}}$ = 0 $\overline{\texttt{ABE[0]}}$ = 1	$\overline{\texttt{ABE[1]}}$ = 0 $\overline{\texttt{ABE[0]}}$ = 0 (invalid)

AMC Programming Model

The asynchronous memory global control register (EBIU_AMGCTL) configures global aspects of the controller. It contains bank enables and other information as described in this section. This register should not be programmed while the AMC is in use. The EBIU_AMGCTL register should be

the last control register written to when configuring the processor to access external memory-mapped asynchronous devices.

Additional information for the EBIU_AMGCTL register bits includes:

- Asynchronous memory clock enable (AMCKEN)

  For external devices that need a clock, CLKOUT can be enabled by setting the AMCKEN bit in the EBIU_AMGCTL register. In systems that do not use CLKOUT, set the AMCKEN bit to 0.

- Asynchronous memory bank enable (AMBEN).

  If a bus operation accesses a disabled asynchronous memory bank, the EBIU responds by acknowledging the transfer and asserting the error signal on the requesting bus. The error signal propagates back to the requesting bus master. This generates a hardware exception to the core, if it is the requester. For DMA mastered requests, the error is captured in the respective status register. If a bank is not fully populated with memory, then the memory likely aliases into multiple address regions within the bank. This aliasing condition is not detected by the EBIU, and no error response is asserted.

- Core/DMA priority (CDPRIO).

  This bit configures the AMC to control the priority over requests that occur simultaneously to the EBIU from either processor core or the DMA controller. When this bit is set to 0, a request from the core has priority over a request from the DMA controller to the AMC, unless the DMA is urgent. When the CDPRIO bit is set, all requests from the DMA controller, including the memory DMAs, have priority over core accesses. For the purposes of this discussion, core accesses include both data fetches and instruction fetches.

The CDPRIO bit also applies to the SDC.

The EBIU asynchronous memory controller has two asynchronous memory bank control registers (EBIU_AMBCTLO and EBIU_AMBCTL1). They

contain bits for counters for setup, access, and hold time; bits to determine memory type and size; and bits to configure use of ARDY. These registers should not be programmed while the AMC is in use.

The timing characteristics of the AMC can be programmed using these four parameters:

- Setup: the time between the beginning of a memory cycle ($\overline{\text{AMS[x]}}$ low) and the read-enable assertion ($\overline{\text{ARE}}$ low) or write-enable assertion ($\overline{\text{AWE}}$ low).
- Read access: the time between read-enable assertion ($\overline{\text{ARE}}$ low) and deassertion ($\overline{\text{ARE}}$ high).
- Write access: the time between write-enable assertion ($\overline{\text{AWE}}$ low) and deassertion ($\overline{\text{AWE}}$ high).
- Hold: the time between read-enable deassertion ($\overline{\text{ARE}}$ high) or write-enable deassertion ($\overline{\text{AWE}}$ high) and the end of the memory cycle ($\overline{\text{AMS[x]}}$ high).

Each of these parameters can be programmed in terms of EBIU clock cycles. In addition, there are minimum values for these parameters:

- Setup ≥ 1 cycle
- Read access ≥ 1 cycle
- Write access ≥ 1 cycle
- Hold ≥ 0 cycles

AMC Registers

The following sections describe the AMC registers.

EBIU_AMGCTL Register

Figure 7-6 shows the asynchronous memory global control register (EBIU_AMGCTL).

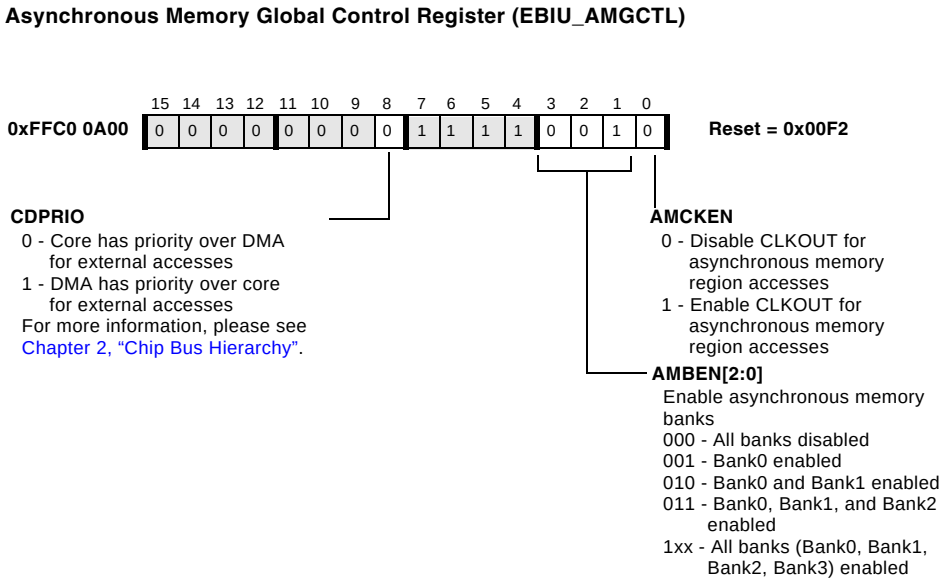

Asynchronous Memory Global Control Register (EBIU_AMGCTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0A00
0 0 0 0 0 0 0 0 1 1 1 1 0 0 1 0
Reset = 0x00F2
CDPRIO
0 - Core has priority over DMA for external accesses
1 - DMA has priority over core for external accesses
For more information, please see Chapter 2, "Chip Bus Hierarchy".
AMCKEN
0 - Disable CLKOUT for asynchronous memory region accesses
1 - Enable CLKOUT for asynchronous memory region accesses
AMBEN[2:0]
Enable asynchronous memory banks
000 - All banks disabled
001 - Bank0 enabled
010 - Bank0 and Bank1 enabled
011 - Bank0, Bank1, and Bank2 enabled
1xx - All banks (Bank0, Bank1, Bank2, Bank3) enabled

Figure 7-6. Asynchronous Memory Global Control Register

EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers

Figure 7-7 and Figure 7-8 show the asynchronous memory bank control registers (EBIU_AMBCTL0 and EBIU_AMBCTL1).

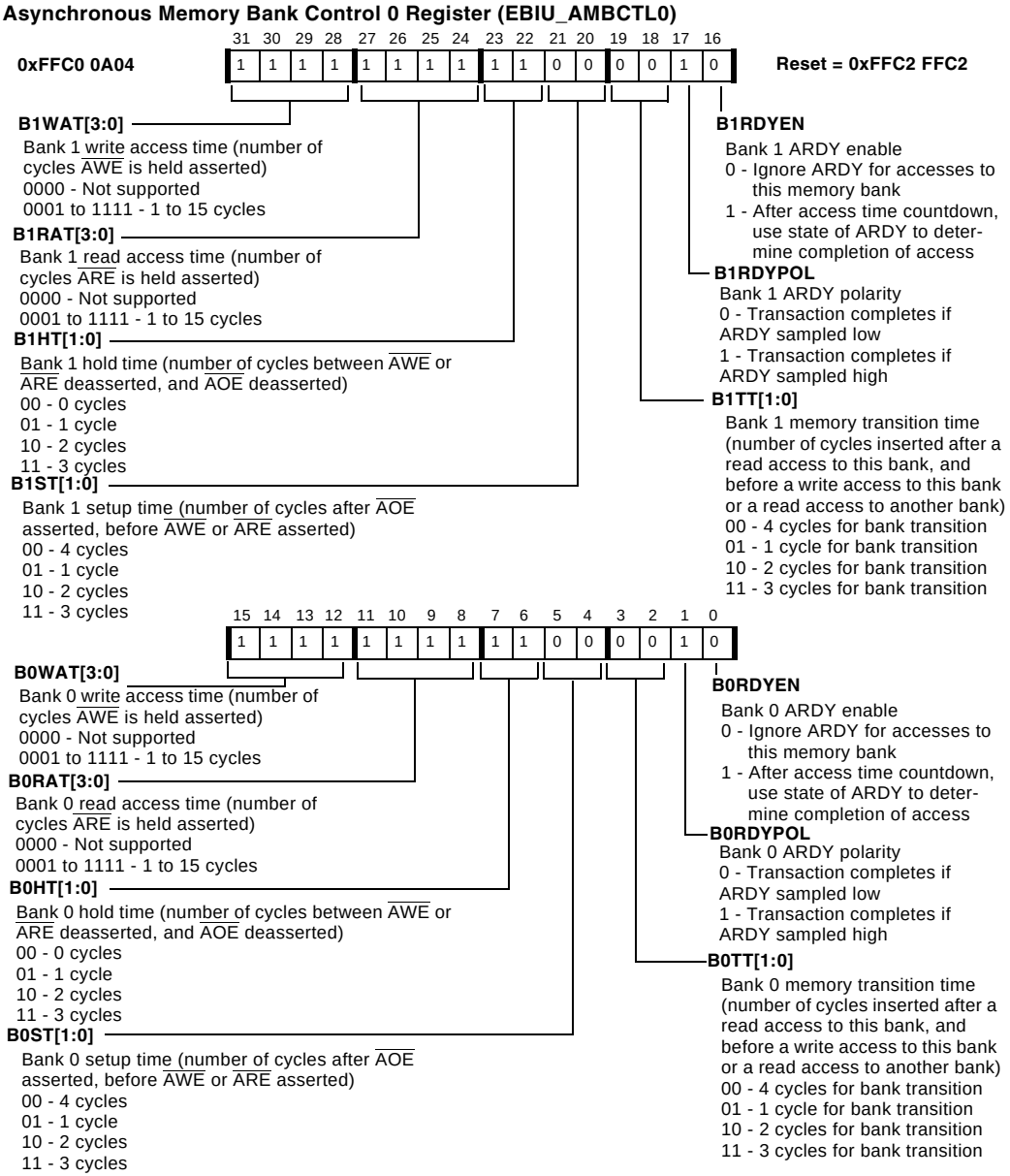

Asynchronous Memory Bank Control 0 Register (EBIU_AMBCTL0)
0xFFC0 0A04
Reset = 0xFFC2 FFC2
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 0
B1WAT[3:0]
Bank 1 write access time (number of cycles AWE is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles
B1RAT[3:0]
Bank 1 read access time (number of cycles ARE is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles
B1HT[1:0]
Bank 1 hold time (number of cycles between AWE or ARE deasserted, and AOE deasserted)
00 - 0 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles
B1ST[1:0]
Bank 1 setup time (number of cycles after AOE asserted, before AWE or ARE asserted)
00 - 4 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles
B1RDYEN
Bank 1 ARDY enable
0 - Ignore ARDY for accesses to this memory bank
1 - After access time countdown, use state of ARDY to determine completion of access
B1RDYPOL
Bank 1 ARDY polarity
0 - Transaction completes if ARDY sampled low
1 - Transaction completes if ARDY sampled high
B1TT[1:0]
Bank 1 memory transition time (number of cycles inserted after a read access to this bank, and before a write access to this bank or a read access to another bank)
00 - 4 cycles for bank transition
01 - 1 cycle for bank transition
10 - 2 cycles for bank transition
11 - 3 cycles for bank transition
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 0
B0WAT[3:0]
Bank 0 write access time (number of cycles AWE is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles
B0RAT[3:0]
Bank 0 read access time (number of cycles ARE is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles
B0HT[1:0]
Bank 0 hold time (number of cycles between AWE or ARE deasserted, and AOE deasserted)
00 - 0 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles
B0ST[1:0]
Bank 0 setup time (number of cycles after AOE asserted, before AWE or ARE asserted)
00 - 4 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles
B0RDYEN
Bank 0 ARDY enable
0 - Ignore ARDY for accesses to this memory bank
1 - After access time countdown, use state of ARDY to determine completion of access
B0RDYPOL
Bank 0 ARDY polarity
0 - Transaction completes if ARDY sampled low
1 - Transaction completes if ARDY sampled high
B0TT[1:0]
Bank 0 memory transition time (number of cycles inserted after a read access to this bank, and before a write access to this bank or a read access to another bank)
00 - 4 cycles for bank transition
01 - 1 cycle for bank transition
10 - 2 cycles for bank transition
11 - 3 cycles for bank transition

Figure 7-7. Asynchronous Memory Bank Control 0 Register

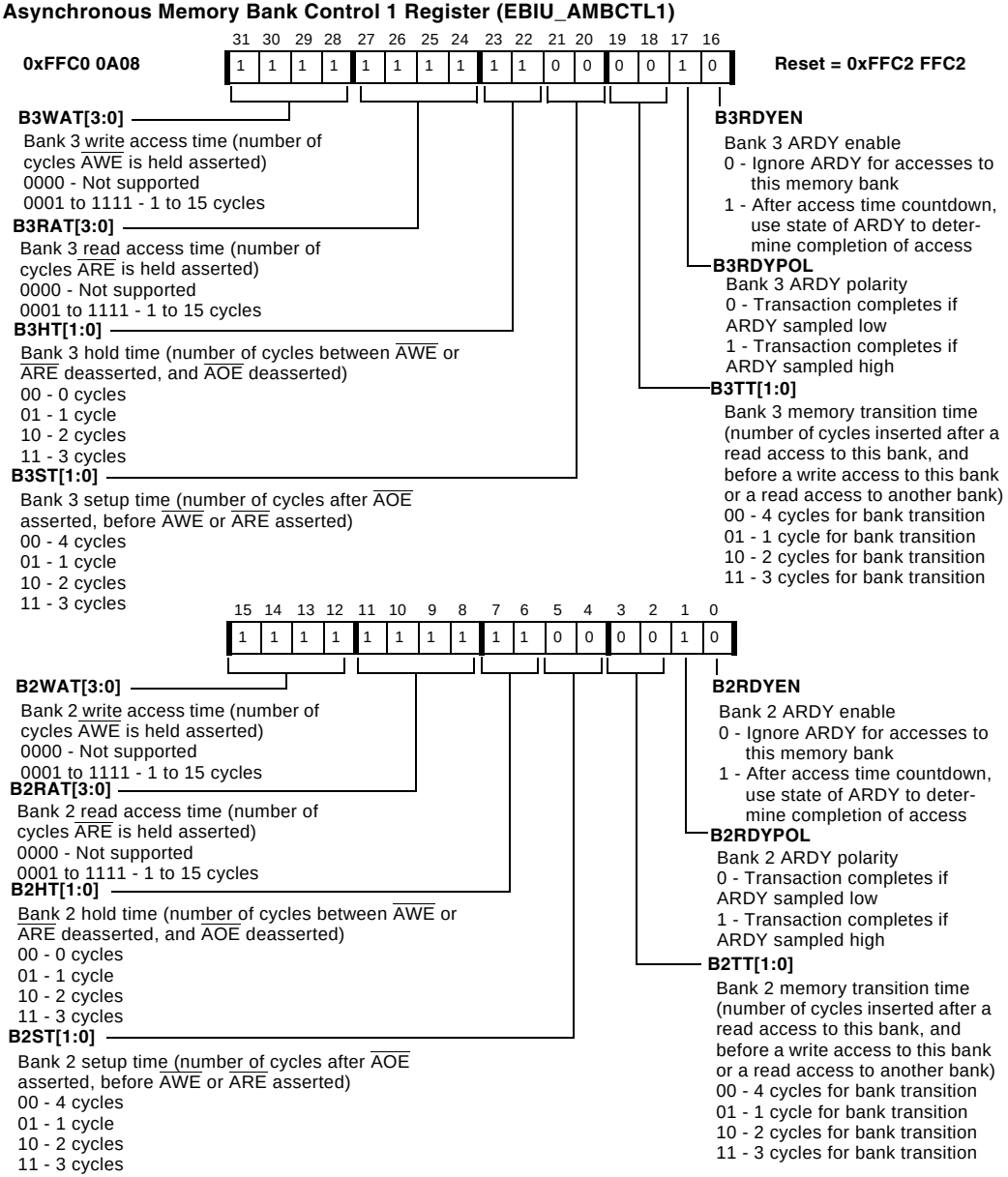

Asynchronous Memory Bank Control 1 Register (EBIU_AMBCTL1)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0xFFC0 0A08
1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 0
Reset = 0xFFC2 FFC2
B3WAT[3:0]
Bank 3 write access time (number of cycles AWE is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles
B3RAT[3:0]
Bank 3 read access time (number of cycles ARE is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles
B3HT[1:0]
Bank 3 hold time (number of cycles between AWE or ARE deasserted, and AOE deasserted)
00 - 0 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles
B3ST[1:0]
Bank 3 setup time (number of cycles after AOE asserted, before AWE or ARE asserted)
00 - 4 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles
B3RDYEN
Bank 3 ARDY enable
0 - Ignore ARDY for accesses to this memory bank
1 - After access time countdown, use state of ARDY to determine completion of access
B3RDYPOL
Bank 3 ARDY polarity
0 - Transaction completes if ARDY sampled low
1 - Transaction completes if ARDY sampled high
B3TT[1:0]
Bank 3 memory transition time (number of cycles inserted after a read access to this bank, and before a write access to this bank or a read access to another bank)
00 - 4 cycles for bank transition
01 - 1 cycle for bank transition
10 - 2 cycles for bank transition
11 - 3 cycles for bank transition
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 0
B2WAT[3:0]
Bank 2 write access time (number of cycles AWE is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles
B2RAT[3:0]
Bank 2 read access time (number of cycles ARE is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles
B2HT[1:0]
Bank 2 hold time (number of cycles between AWE or ARE deasserted, and AOE deasserted)
00 - 0 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles
B2ST[1:0]
Bank 2 setup time (number of cycles after AOE asserted, before AWE or ARE asserted)
00 - 4 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles
B2RDYEN
Bank 2 ARDY enable
0 - Ignore ARDY for accesses to this memory bank
1 - After access time countdown, use state of ARDY to determine completion of access
B2RDYPOL
Bank 2 ARDY polarity
0 - Transaction completes if ARDY sampled low
1 - Transaction completes if ARDY sampled high
B2TT[1:0]
Bank 2 memory transition time (number of cycles inserted after a read access to this bank, and before a write access to this bank or a read access to another bank)
00 - 4 cycles for bank transition
01 - 1 cycle for bank transition
10 - 2 cycles for bank transition
11 - 3 cycles for bank transition

Figure 7-8. Asynchronous Memory Bank Control 1 Register

AMC Programming Examples

Listing 7-1 and Listing 7-2 provide examples for working with the AMC.

Listing 7-1. 16-Bit Core Transfers to SRAM

```
.section L1_data_b;
.byte2 source[N] = 0x1122, 0x3344, 0x5566, 0x7788;
.section SRAM_bank_0;
.byte2 dest[N];
.section L1_code;
I0.L = lo(source);
I0.H = hi(source);
I1.L = lo(dest);
I1.H = hi(dest);
     R0.L = w[I0++];
     P5=N-1;
     lsetup(lp, lp) LC0=P5;
lp:  R0.L = w[I0++] || w[I1++] = R0.L;
                       w[I1++] = R0.L;
```

Listing 7-2. 8-Bit Core Transfers to SRAM Using Byte Mask ABE[1:0] Pins

```
.section L1_data_b;
.byte source[N] = 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88;
.section SRAM_bank_0;
.byte dest[N];
p0.L = lo(source);
p0.H = hi(source);
p1.L = lo(dest);
p1.H = hi(dest);
```

```
        p5=N;
        lsetup(start, end) LC0=P5;
start:  R0 = b[p0++](z);
end:    b[p1++] = R0;   /* byte data masking */
```

SDC Overview and Features

The SDRAM Controller (SDC) enables the processor to transfer data to and from Synchronous DRAM (SDRAM) with a maximum frequency specified in the product data sheet. The processor supports a glueless interface with one external bank of standard SDRAMs of 64M bit to 512M bit, with configurations x4, x8, and x16, up to a maximum total capacity of 128M bytes of SDRAM.

Features

The EBIU SDC provides a glueless interface with standard SDRAMs. Features include:

- I/O width 16-bit, I/O supply 1.8, 2.5 or 3.3 V
- Supports up to 128M byte of SDRAM in external bank
- Types of 64, 128, 256, and 512M bit with I/O of x4, x8, and x16
- Supports SDRAM page sizes of 512 byte, 1K, 2K, and 4K byte
- Supports multibank operation within the SDRAM
- Supports mobile SDRAMs
- SDC uses no-burst mode (`BL = 1`) with sequential burst type
- SDC supports 8-bit data masking writes
- SDC uses open page policy—any open page is closed only if a new access in another page of the same bank occurs

- Uses a programmable refresh counter to coordinate between varying clock frequencies and the SDRAM's required refresh rate
- Provides multiple timing options to support additional buffers between the processor and SDRAM
- Allows independent auto-refresh while the asynchronous memory controller has control of the EBIU port
- Supports self-refresh mode for power savings
- During hibernate state, self-refresh mode is supported
- Supports instruction fetch

SDRAM Configurations Supported

Table 7-4 shows all possible bank sizes, and SDRAM discrete component configurations that can be gluelessly interfaced to the SDC. The bank width for all cases is 16 bits.

Table 7-4. SDRAM Discrete Component Configurations Supported

System Size (M byte)	System Size (M bit)	SDRAM Configuration	Number of Chips
16	8M x 16	8M x 8	2
16	8M x 16	8M x 16	1
32	16M x 16	16M x 4	4
32	16M x 16	16M x 8	2
32	16M x 16	16M x 16	1
64	32M x 16	32M x 4	4
64	32M x 16	32M x 8	2
64	32M x 16	32M x 16	1
128	64M x 16	64M x 4	4

Table 7-4. SDRAM Discrete Component Configurations Supported (Continued)

System Size (M byte)	System Size (M bit)	SDRAM Configuration	Number of Chips
128	64M x 16	64M x 8	2
128	64M x 16	64M x 16	1

SDRAM External Bank Size

The total amount of external SDRAM memory addressed by the processor is controlled by the EBSZ bits of the EBIU_SDBCTL register (see Table 7-5). Accesses above the range shown for a specialized EBSZ value results in an internal bus error and the access does not occur. For more information, see “Error Detection” on page 7-7.

SDC Address Mapping

The address mapping scheme describes how the SDC maps the address into SDRAM. To access SDRAM, the SDC uses the bank interleaving map scheme, which fills each internal SDRAM bank before switching to the next internal bank. Since the SDRAMs have four internal banks, the entire SDRAM address space is therefore divided into four sub-address regions containing the addresses of each internal bank. (See Figure 7-10 on page 7-39.) It starts with address 0x0 for internal bank A and ends with the last valid address (specified with EBSZ and EBCAW parameters) containing the internal bank D.

The internal 29-bit non-multiplexed address (See Figure 7-9) is multiplexed into:

- Byte data mask (IA[0])
- SDRAM column address

- SDRAM row address
- Internal SDRAM bank address

A good understanding of the SDC address map scheme in conjunction with the multibank operation is required to obtain optimized system performance.

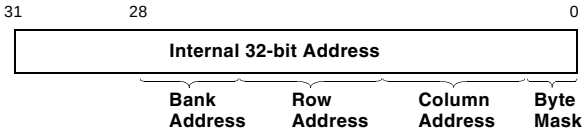


Figure 7-9. Multiplexed SDRAM Addressing Scheme

Table 7-5. External Bank Size Encodings

EBSZ	Bank Size (M byte)	Valid SDRAM Addresses
000	16	0x0000 0000 – 0x00FF FFFF
001	32	0x0000 0000 – 0x01FF FFFF
010	64	0x0000 0000 – 0x03FF FFFF
011	128	0x0000 0000 – 0x07FF FFFF

Internal SDRAM Bank Select

The internal SDRAM banks are driven by the ADSP-BF52x `ADDR[19:18]` which are part of the row and column address and connected to the SDRAM `BA[1:0]`.

Do not flip up both internal bank select connections, if using the mobile SDRAM PASR feature. If this is done, the system will not work properly because the selected internal banks are not refreshed during partial array self-refresh.

Parallel Connection of SDRAMs

To specify an SDRAM system, multiple possibilities are given based on the different architectures. (See Table 7-13 on page 7-63.) For the ADSP-BF52x processors, I/O capabilities of 1 x 16-bit, 2 x 8-bit or 4 x 4-bit are given. The reason to use a system of 4 x 4-bit vs. 2 x 8-bit or 1 x 16-bit is determined by the SDRAM page size. All 3 systems have the same external bank size, but different page sizes. On one hand, the higher the page size, the higher the performance. On the other hand, the higher the page size, the higher the hardware layout requirements.

Even if connecting SDRAMs in parallel, the SDC always considers the entire system as one external SDRAM bank (`SMS` pin) because all address and control lines feed the parallel parts.

However, access to a single cluster part is achieved using the mask feature (`SDQM[1:0]` pins). This allows masked 8-bit I/O writes to dedicated chips whereby the other 8-bit I/O is masked at its input buffer of the other chips. See Listing 7-4 on page 7-77.

SDC Interface Overview

The following sections describe the SDC interface.

SDC Pin Description

The SDRAM interface signals are shown in Table 7-6.

Table 7-6. SDRAM Interface Signals

Pad	Pin Type [1]	Description
DATA[15:0]	I/O	External data bus
ADDR[19:18], ADDR[16:12], ADDR[10:1]	O	External address bus Connect to SDRAM address pins. Bank address is output on ADDR[19:18] and should be connected to SDRAM BA[1:0] pins.
$\overline{\text{SRAS}}$	O	SDRAM row address strobe pin Connect to SDRAM's $\overline{\text{RAS}}$ pin.
$\overline{\text{SCAS}}$	O	SDRAM column address strobe pin Connect to SDRAM's $\overline{\text{CAS}}$ pin.
$\overline{\text{SWE}}$	O	SDRAM write enable pin Connect to SDRAM's $\overline{\text{WE}}$ pin.
$\overline{\text{ABE[1:0]}}$/ SDQM[1:0]	O	SDRAM data mask pins Connect to SDRAM's DQM pins.
$\overline{\text{SMS}}$	O	Memory select pin of external memory bank configured for SDRAM Connect to SDRAM's $\overline{\text{CS}}$ (Chip Select) pin. Active low.
SA10	O	SDRAM A10 pin SDRAM interface uses this pin to be able to do refreshes while the AMC is using the bus. Connect to SDRAM's A[10] pin.
SCKE	O	SDRAM clock enable pin Connect to SDRAM's CKE pin.
CLKOUT	O	SDRAM clock output pin Switches at system clock frequency. Connect to the SDRAM's CLK pin.

1 Pin Types: I = Input, O = Output

SDRAM Performance

On-page sequential or non-sequential accesses are from internal data memory to SDRAM. Table 7-7 summarizes SDRAM performance for these on-page accesses.

Table 7-7. SDRAM Performance Between Internal Data Memory and SDRAM[1]

Type of access	Performance
DAG access, write	1 SCLK cycle per 16-bit word
DAG access, read	8 SCLK cycles per 16-bit word
MemDMA access, write	1 SCLK cycle per 16-bit word
MemDMA access, read	≈1.1 SCLK cycles per 16-bit word

1 Valid for core/system clock > 2:1

On-page sequential instruction fetches from SDRAM are summarized in Table 7-8.

Table 7-8. SDRAM Performance For On-Page Instruction Fetches

Type of access	Performance
Ifetch from SDRAM	≈1.1 SCLK cycles per 16-bit word
I/Dcache line fill from SDRAM	≈1.1 SCLK cycles per 16-bit word

Off-page accesses are summarized in Table 7-9.

Table 7-9. SDRAM Stall Cycles For Off-Page Accesses

Type of access	Stall Cycles
Write	$t_{WR} + t_{RP} + t_{RCD}$
Read	$t_{RP} + t_{RCD} + CL$

SDC Description of Operation

The following sections describe the operation of the SDC.

Definition of SDRAM Architecture Terms

The following are definitions of SDRAM architecture terms used in the remainder of this chapter.

Refresh

Since the information is stored in a low-capacitance cell that suffers from leakage effects, the SDRAM must be refreshed periodically.

Row Activation

SDRAM accesses are multiplexed, which means any first access will open a row/page before the column access is performed. It stores the row in a "row cache" called row activation.

Column Read/Write

The row's columns represent a page, which can be accessed with successive read or write commands without needing to activate another row. This is called column access and performs transfers from the "row cache."

Row Precharge

If the next access is in a different row, the current row is closed before another is opened. The current "row cache" is written back to the row. This is called row precharge.

Internal Bank

There are up to 4 internal memory banks on a given SDRAM. Each of these banks can be accessed with the bank select lines BA[1:0]. The bank address can be thought of as part of the row address.

External Bank

This is the address region where the SDC address the SDRAM.

Do not confuse the internal banks, which are internal to the SDRAM and are selected with the BA[1:0] pins with the external bank that is enabled by the $\overline{\text{CS}}$ pin.

Memory Size

Since the 2-D memory is based on rows and columns, the size is:

```
mem size =
         (# rows) x (# columns) x (# internal banks) x I/O (Mbit)
```

Burst Length

The burst length determines the number of words that the SDRAM device stores or delivers after detecting a single write or read command followed by a NOP (no operation) command, respectively (Number of NOPs = burst length - 1). Burst lengths of full page, 8, 4, 2, and 1 (no burst) are available. The burst length is selected by writing the BL bits in the SDRAM mode register during the SDRAM powerup sequence.

Burst Type

The burst type determines the address order in which the SDRAM delivers burst data. The burst type is selected by writing the BT bits in the SDRAM mode register during the SDRAM powerup sequence.

CAS Latency

The CAS latency, or read latency, specifies the time between latching a read address and driving the data off chip. This specification is normalized to the system clock and varies from 2 to 3 cycles based on the speed. The CAS latency is selected by writing the CL bits in the SDRAM mode register during the SDRAM powerup sequence.

Data I/O Mask Function

SDRAMs allow a data byte-masking capability on writes. The DQM[1:0] mask pins are used to block the data input buffer of the SDRAM during write operations.

SDRAM Commands

SDRAM commands are not based on typical read or write strobes. The pulsed $\overline{\text{CS}}$, $\overline{\text{RAS}}$, $\overline{\text{CAS}}$, and $\overline{\text{WE}}$ lines determine the command on the rising clock edge by a truth table.

Mode Register Set (MRS) command

SDRAM devices contain an internal extended configuration register which allows specification of the mobile SDRAM device's functionality.

Extended Mode Register Set (EMRS) command

Mobile SDRAM devices contain an internal extended configuration register which allows specification of the mobile SDRAM device's functionality.

Bank Activate command

The bank activate command causes the SDRAM to open an internal bank (specified by the bank address) in a row (specified by the row address). When the bank activate command is issued, it opens a new row address in

the dedicated bank. The memory in the open internal bank and row is referred to as the open page. The bank activate command must be applied before a read or write command.

Read/Write command

For the read command, the SDRAM latches the column address. The start address is set according to the column address. For the write command, SDRAM latches the column address. Data is also asserted in the same cycle. The start address is set according to the column address.

Precharge/Precharge All Command

The precharge command closes a specific active page in an internal bank and the precharge all command closes all 4 active pages in all 4 banks.

Auto-refresh command

When the SDC refresh counter times out, the SDC precharges all four banks of SDRAM and then issues an auto-refresh command to them. This causes the SDRAM to generate an internal auto-refresh cycle. When the internal refresh completes, all four internal SDRAM banks are precharged.

Enter Self-Refresh Mode

When the SDRAM enters self-refresh mode, the SDRAM's internal timer initiates refresh cycles periodically, without external control input.

Exit Self-Refresh Mode

When the SDRAM exits self-refresh mode, the SDRAM's internal timer stops refresh cycles and relinquishes control to external SDC.

SDC Timing Specifications

The following SDRAM timing specifications are used by the SDC and SDRAM. To program the SDRAM interface, see the SDRAM specific datasheet information

Any absolute timing parameter must be normalized to the system clock, which allows the SDC to adapt to the timing parameter of the device.

t_{MRD}

This is the required delay between issuing a mode register set and an activate command during powerup.

Dependency: system clock frequency
SDC setting: 3 system clock cycles
SDC usage: MRS command

t_{RAS}

This is the required delay between issuing a bank A activate command and issuing a bank A precharge command.

Dependency: system clock frequency
SDC setting: 1–15 normalized system clock cycles
SDC usage: single column read/write, auto-refresh, self-refresh command

t_{CL}

The CAS latency, or read latency, is the delay between when the SDRAM detects the read command and when it provides the data off-chip. This specification does not apply to writes.

Dependency: system clock frequency and speed grade
SDC setting: 2–3 normalized system clock cycles
SDC usage: first read command

t_{RCD}

This is the required delay between a bank A activate command and the first bank A read or write command.

Dependency: system clock frequency
SDC setting: 1–7 normalized system clock cycles
SDC usage: first read/write command

t_{RRD}

This is the required delay between a bank A activate command and a bank B activate command. This specification is used for multibank operation.

Dependency: system clock frequency
SDC setting: t_{RCD} + 1 normalized system clock cycles
SDC usage: multiple bank activation

t_{WR}

This is the required delay between a bank A write command and a bank A precharge command. This specification does not apply to reads.

Dependency: system clock frequency
SDC setting: 1–3 normalized system clock cycles
SDC usage: during off-page write command

t_{RP}

This is the required delay between a bank A precharge command and a bank A activation command.

Dependency: system clock frequency
SDC setting: 1–7 normalized system clock cycles
SDC usage: off-page read/write, auto-refresh, self-refresh command

t_{RC}

This is the required delay between issuing successive bank activate commands.

Dependency: system clock frequency
SDC setting: user must ensure that $t_{RP} + t_{RAS} \geq \max(t_{RC}, t_{RFC}, t_{XSR})$
SDC usage: single column read/write command

t_{RFC}

This is the required delay between issuing successive auto-refresh commands (all banks).

Dependency: system clock frequency
SDC setting: user must ensure that $t_{RP} + t_{RAS} \geq \max(t_{RC}, t_{RFC}, t_{XSR})$
SDC usage: auto-refresh, exit self-refresh command

t_{XSR}

This is the required delay between exiting self-refresh mode and the auto-refresh command.

Dependency: system clock frequency
SDC setting: user must ensure that $t_{RP} + t_{RAS} \geq \max(t_{RC}, t_{RFC}, t_{XSR})$
SDC usage: exit self-refresh command

t_{REF}

This is the row refresh period, and typically takes 64 ms.

Dependency: system clock frequency
SDC setting: none
SDC usage: auto-refresh command

t_{REFI}

This is the row refresh interval and typically takes 15.6 µs for < 8k rows and 7.8 µs for ≥ 8k rows. This specification is available by dividing t_{REF} by the number of rows. This number is used by the SDC refresh counter.

Dependency: system clock frequency
SDC setting: t_{REFI} normalized system clock cycles (RDIV register)
SDC usage: auto-refresh command

In typical applications making sequential (not random) accesses to the SDRAM memory, the t_{RAS} timing parameter is less critical than t_{RP}. Be aware that whenever the sum of t_{RP} + t_{RAS} is violating one of the other timing specifications, the t_{RAS} parameter should be increased.

SDC Functional Description

The functional description of the SDC is provided in the following sections.

SDC Operation

The AMC normally generates an external memory address, which then asserts the corresponding CS select, along with RD and WR strobes. However these control signals are not used by the SDC. The internal strobes are used to generate pulsed commands ($\overline{\text{SMS}}$, SCKE, $\overline{\text{SRAS}}$, $\overline{\text{SCAS}}$, $\overline{\text{SWE}}$) within a truth table (see Table 7-11 on page 7-46). The memory access to SDRAM is based by mapping ADDR[28:0] causing an internal memory select to SDRAM space (see Figure 7-10).

The configuration is programmed in the SDBCTL register. The SDRAM controller can hold off the processor core or DMA controller with an internally connected acknowledge signal, as controlled by refresh, or page miss latency overhead.

A programmable refresh counter is provided which generates background auto-refresh cycles at the required refresh rate based on the clock frequency used. The refresh counter period is specified with the RDIV field in the SDRAM refresh rate control register.

To allow auto-refresh commands to execute in parallel with any AMC access, a separate A10 pin (SA10) is provided.

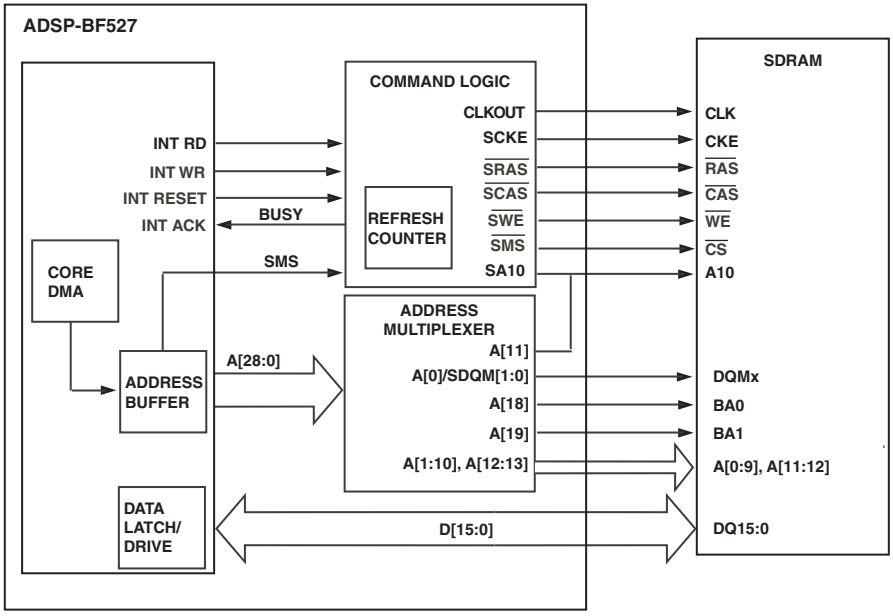


Figure 7-10. Simplified SDC Architecture

The internal 32-bit non-multiplexed address is multiplexed into:

- Data mask for bytes
- SDRAM column address
- SDRAM row address
- Internal SDRAM bank address

Bit A[0] is used for 8-bit wide SDRAMs to generate the data masks. The next lowest bits are mapped into the column address, next bits are mapped into the row address, and the final two bits are mapped into the internal bank address. This mapping is based on the EBCAW and EBSZ values programmed into the SDRAM memory bank control register.

The SDC uses no burst mode (BL = 1) for read and write operations. This requires the SDC to post every read or write address on the bus as for non-sequential reads or writes, but does not cause any performance degradation. For read commands, there is a latency from the start of the read command to the availability of data from the SDRAM, equal to the CAS latency. This latency is always present for any single read transfer. Subsequent reads do not have latency.

Whenever a page miss to the same bank occurs, the SDC executes a precharge command followed by a bank activate command before executing the read or write command. If there is a page hit, the read or write command can be given immediately without requiring the precharge command.

SDC Address Muxing

Table 7-10 shows the connection of the address pins with the SDRAM device pins.

Table 7-10. SDRAM Address Connections for 16-bit Banks

External Address Pin	SDRAM Address Pin
ADDR[19]	BA[1]
ADDR[18]	BA[0]
ADDR[16]	A[15]
ADDR[15]	A[14]
ADDR[14]	A[13]
ADDR[13]	A[12]
ADDR[12]	A[11]
ADDR[11]	Not used
SA[10]	A[10]
ADDR[10]	A[9]
ADDR[9]	A[8]
ADDR[8]	A[7]
ADDR[7]	A[6]
ADDR[6]	A[5]
ADDR[5]	A[4]
ADDR[4]	A[3]
ADDR[3]	A[2]
ADDR[2]	A[1]
ADDR[1]	A[0]

Multibank Operation

Since every SDRAM chip contains 4 independent internal banks (A-D), the SDC is capable of supporting multibank operation thus taking advantage of the architecture.

Any first access to SDRAM bank (A) will force an activate command before a read or write command. However, if any new access falls into the address space of the other banks (B, C, D) the SDC leaves bank (A) open and activates any of the other banks (B, C, D). Bank (A) to bank (B) active time is controlled by $t_{RRD} = t_{RCD} + 1$. This scenario is repeated until all 4 banks (A-D) are opened and results in an effective page size up to 4 pages because no latency causes switching between these open pages (compared to 1 page in only one bank at the time). Any access to any closed page in any opened bank (A-D) forces a precharge command only to that bank. If, for example, 2 MemDMA channels are pointing to the same internal SDRAM bank, this always forces precharge and activation cycles to switch between the different pages. However, if the 2 MemDMA

channels are pointing to different internal SDRAM banks, it does not cause additional overhead. See Figure 7-11.

The benefit of multibank operation reduces precharge and activation cycles by mapping opcode/data among different internal SDRAM banks driven by the `A[19:18]` pins.

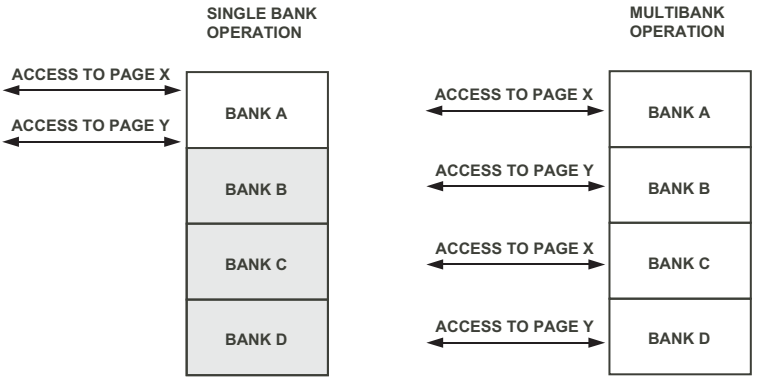


Figure 7-11. SDRAM Bank Operation Types

Core and DMA Arbitration

The `CDPRIO` bit configures the SDC to control the priority over requests that occur simultaneously to the EBIU from either the processor core or the DMA controller. When this bit is set to 0, a request from the core has priority over a request from the DMA controller to the SDC, unless the DMA is urgent. When it is set to 1, all requests from the DMA controller, including the memory DMAs, have priority over core accesses. For the purposes of this discussion, core accesses include both data fetches and instruction fetches. For additional information see "Using the CDPRIO Bit to Change Priorities" on page 2-13.

Changing System Clock During Runtime

All timing specifications are normalized to the system clock. Since most of them are minimum specifications, except t_{REF}, which is a maximum specification, a variation of system clock will on one hand violate a specific specification and on the other hand cause a performance degradation for the other specifications.

The reduction of system clock will violate the minimum specifications, while increasing system clock will violate the maximum t_{REF} specification. Therefore, careful software control is required to adapt these changes.

For most applications, the SDRAM powerup sequence and writing of the mode register needs to be done only once. Once the powerup sequence has completed, the PSSE bit should not be set again unless a change to the mode register is desired.

The recommended procedure for changing the PLL VCO frequency is:

1. Issue an SSYNC instruction to ensure all pending memory operations have completed.
2. Set the SDRAM to self-refresh mode by writing a 1 to the SRFS bit of EBIU_SDGCTL.
3. Execute the desired PLL programming sequence. (For details, refer to Chapter 18, “Dynamic Power Management”.)
4. After the wakeup occurs that signifies the PLL has settled to the new VCO frequency, reprogram the SDRAM registers (EBIU_SDRRC, EBIU_SDGCTL) with values appropriate to the new SCLK frequency, and assure that the PSSE bit is set.
5. Bring the SDRAM out of self-refresh mode by clearing the SRFS bit of EBIU_SDGCTL. The SDRAM will stay within the self-refresh until the first access.

Changing the SCLK frequency using the SSEL bits in PLL_DIV, as opposed to actually changing the VCO frequency, should be done using these steps:

1. Issue an SSYNC instruction to ensure all pending memory operations have completed.
2. Set the SDRAM to self-refresh mode by writing a 1 to the SRFS bit of EBIU_SDGCTL.
3. Execute the desired write to the SSEL bits.
4. Reprogram the SDRAM registers with values appropriate to the new SCLK frequency, and assure that the PSSE bit is set.
5. Bring the SDRAM out of self-refresh mode by clearing the SRFS bit of EBIU_SDGCTL.

Changing Power Management During Runtime

Deep Sleep Mode

During deep sleep mode, the core and system clock will halt. Therefore, a careful software control is required to enter SDRAM in self-refresh before the device enters deep sleep mode.

Hibernate State

In the hibernate state the core voltage is 0 (core reset), but the I/O voltage can still be applied. In order to save the SDRAM volatile data, the ADSP-BF52x processor supports driving the SCKE signal low during core reset. Setting the SCKELOW bit of VR_CTL keeps the SCKE signal low. This ensures that the self-refresh mode is not exited during the reset sequence initiated by a hibernate wake-up event. Normally, the SCKE pin is toggled high during reset to comply with PC-133 specifications. For details about the SCKELOW bit, refer to Chapter 18, “Dynamic Power Management”.

SDC Commands

This section provides a description of each of the commands that the SDC uses to manage the SDRAM interface. These commands are initiated automatically upon a memory read or memory write. A summary of the various commands used by the on-chip controller for the SDRAM interface is as follows.

- MODE REGISTER SET
- EXTENDED MODE REGISTER SET
- BANK ACTIVATION
- READ and WRITE
- SINGLE PRECHARGE
- PRECHARGE ALL
- AUTO-REFRESH
- SELF-REFRESH ENTRY and SELF-REFRESH EXIT
- NOP

Table 7-11 shows the SDRAM pin state during SDC commands.

Table 7-11. Pin State During SDC Commands

Command	SCKE (n - 1)	SCKE (n)	$\overline{\text{SMS}}$	$\overline{\text{SRAS}}$	$\overline{\text{SCAS}}$	$\overline{\text{SWE}}$	SA10	Addresses
(E)/Mode register set	High	High	Low	Low	Low	Low	Op-code	Op-code
Activate	High	High	Low	Low	High	High	Valid address bit	Valid
Read	High	High	Low	High	Low	High	Low (CMD)	Valid

Table 7-11. Pin State During SDC Commands (Continued)

Command	SCKE (n - 1)	SCKE (n)	$\overline{\text{SMS}}$	$\overline{\text{SRAS}}$	$\overline{\text{SCAS}}$	$\overline{\text{SWE}}$	SA10	Addresses
Single precharge	High	High	Low	Low	High	Low	Low	Valid
Precharge all	High	High	Low	Low	High	Low	High	Don't care
Write	High	High	Low	High	Low	Low	Low (CMD)	Valid
Auto-refresh	High	High	Low	Low	Low	High	Don't care	Don't care
Self-refresh entry	High	Low	Low	Low	Low	High	Don't care	Don't care
Self-refresh	Low	Low	Don't care	Don't care	Don't care	Don't care	Don't care	Don't care
Self-refresh exit	Low	High	High	Don't care	Don't care	Don't care	Don't care	Don't care
NOP	High	High	Low	High	High	High	Don't care	Don't care
Inhibit	High	High	High	Don't care	Don't Care	Don't care	Don't care	Don't care

Mode Register Set Command

The MODE REGISTER SET (MRS) command initializes SDRAM operation parameters. This command is a part of the SDRAM power-up sequence. The MRS command uses the address bus of the SDRAM as data input. The power-up sequence is initiated by setting the PSSE bit in the SDRAM memory global control register (EBIU_SDGCTL) and then writing or reading from any enabled address within the SDRAM address space to trigger the power-up sequence. The exact order of the power-up sequence is determined by the PSM bit of the EBIU_SDGCTL register.

The MRS command initializes these parameters:

- Burst length = 1, bits A[2-0], always 0
- Burst type = sequential, bit A[3], always 0
- CAS latency, bits A[6-4], programmable in the EBIU_SDGCTL register
- Bits A[12-7], always 0

After power-up and before executing a read or write to the SDRAM memory space, the application must trigger the SDC to write the SDRAM mode register. The write of the SDRAM mode register is triggered by setting the PSSE bit in the SDRAM memory global control register (EBIU_SDGCTL) and then issuing a read or write transfer to the SDRAM address space. The initial read or write triggers the SDRAM power-up sequence to be run, which programs the SDRAM mode register with burst length, burst type, and CAS latency from the EBIU_SDGCTL register and optionally the content to the extended mode register. This initial read or write to SDRAM takes many cycles to complete.

While executing an MRS command, the unused address pins are cleared. During the two clock cycles following the MRS command (t_{MRD}), the SDC issues only NOP commands.

Extended Mode Register Set Command (Mobile SDRAM)

The extended mode register is a subset of the mode register. The EBIU enables programming of the extended mode register during power-up via the EMREN bit in the EBIU_SDGCTL register.

The extended mode register is initialized with these parameters:

- Partial array self-refresh, bits A[2-0], bit A[2] always 0, bits A[1-0] programmable in EBIU_SDGCTL
- Temperature compensated self-refresh, bits A[4-3], bit A[3] always 1, bit A[4] programmable in EBIU_SDGCTL
- Drive strength control, bits A[6-5], always 0
- Bits A[12-7], always 0, and bit A[13] always 1

Not programming the extended mode register upon initialization results in default settings for the low-power features. The extended mode defaults with the temperature sensor enabled, full drive strength, and full array refresh.

Bank Activation Command

The BANK ACTIVATION command is required for first access to any internal bank in SDRAM. Any subsequent access to the same internal bank but different row will be preceded by a precharge and activation command to that bank.

However, if an access to another bank occurs, the SDC leaves the current page open and issues a BANK ACTIVATION command before executing the read or write command to that bank. With this method, called multibank operation, one page per bank can be open at a time, which results in a maximum of four pages.

Read/Write Command

A read/write command is executed if the next read/write access is in the present active page. During the read command, the SDRAM latches the column address. The delay between activate and read commands is determined by the t_{RCD} parameter. Data is available from the SDRAM after the CAS latency has been met.

In the write command, the SDRAM latches the column address. The write data is also valid in the same cycle. The delay between activate and write commands is determined by the t_{RCD} parameter.

The SDC does not use the auto-precharge function of SDRAMs, which is enabled by asserting SA10 high during a read or write command.

Write Command With Data Mask

During partial writes to SDRAM, the SDQM[1:0] pins are used to mask writes to bytes that are not accessed. Table 7-12 shows the SDQM[1:0] encodings based on the internal transfer address bit IA[0] and the transfer size.

During read transfers to SDRAM banks, reads are always done of all bytes in the bank regardless of the transfer size. This means for 16-bit SDRAM banks, SDQM[1:0] are all 0s.

Table 7-12. SDQM[1:0] Encodings During Writes

Internal Address IA[0]	Internal Transfer Size	
	byte	2 bytes
0	SDQM[1] = 1 SDQM[0] = 0	SDQM[1] = 0 SDQM[1] = 0
1	SDQM[0] = 0 SDQM[0] = 1	SDQM[1] = 0 SDQM[1] = 0

For 16-bit SDRAMs, connect SDQM[0] to DQML, and connect SDQM[1] to DQMH.

Single Precharge Command

For a page miss during reads or writes in a specific internal SDRAM bank, the SDC uses the SINGLE PRECHARGE command to that bank.

The SDC does not use the auto-precharge read or write command of SDRAMs, which is enabled by asserting SA10 high during a read or write command.

Precharge All Command

The PRECHARGE ALL command is used to precharge all internal banks at the same time before executing an auto-refresh. All open banks will be automatically closed. This is possible since the SDC uses a separate SA10 pin which is asserted high during this command. This command precedes the AUTO-REFRESH command.

Auto-Refresh Command

The SDRAM internally increments the refresh address counter and causes an auto-refresh to occur internally for that address when the AUTO-REFRESH command is given. The SDC generates an AUTO-REFRESH command after the SDC refresh counter times out. The RDIV value in the SDRAM refresh rate control register must be set so that all addresses are refreshed within the t_{REF} period specified in the SDRAM timing specifications. This command is issued to the external bank whether or not it is enabled (EBE in the SDRAM memory global control register). Before executing the AUTO-REFRESH command, the SDC executes a PRECHARGE ALL command to the external bank. The next activate command is not given until the t_{RFC} specification ($t_{RFC} = t_{RAS} + t_{RP}$) is met.

Auto-refresh commands are also issued by the SDC as part of the powerup sequence and after exiting self-refresh mode.

Self-Refresh Mode

The self-refresh mode is controlled by the SELF-REFRESH ENTRY and SELF-REFRESH EXIT commands. The SDC must issue a series of commands, including the SELF-REFRESH ENTRY command, to put the SDRAM into this low power operation, and it must issue another series of commands, including the SELF-REFRESH EXIT command, to re-access the SDRAM.

Self-Refresh Entry Command

The SELF-REFRESH ENTRY command causes refresh operations to be performed internally by the SDRAM without any external control. This means that the SDC does not generate any auto-refresh commands while the SDRAM is in self-refresh mode. Before executing the SELF-REFRESH ENTRY command, all internal banks are precharged. The SELF-REFRESH ENTRY command is started by setting the SRFS bit of the SDRAM memory global control register (EBIU_SDGCTL). The SDC now drives SCKE low.

Only the SCKE pin keeps control during self-refresh, all other SDRAM pins are allowed to be disabled. However the SDC still drives the SCLK during self-refresh mode. Software may disable the clock by clearing the SCTLE bit in EBIU_SDGCTL.

Self-Refresh Exit Command

Leaving self-refresh mode is performed with the SELF-REFRESH EXIT command, whereby the SDC asserts SCKE. Any internal core/DMA access causes the SDC to perform an SELF-REFRESH EXIT command. The SDC waits to meet the t_{XSR} specification ($t_{XSR} = t_{RAS} + t_{RP}$) and then issues an AUTO-REFRESH command. After the AUTO-REFRESH command, the SDC waits for the t_{RFC} specification ($t_{RFC} = t_{RAS} + t_{RP}$) to be met before executing the activate command for the transfer that caused the SDRAM to exit self-refresh mode. The latency from when a transfer is received by the SDC while in self-refresh mode, until the activate command occurs for that transfer, is:

Time to exit self-refresh: 2 x (t_{RAS} + t_{RP})

The minimum time between a subsequent SELF-REFRESH ENTRY and the SELF-REFRESH EXIT command is at least t_{RAS} cycles. If a self-refresh entry command is issued during any MDMA transfer, the SDC satisfies this core request with the minimum self-refresh period (t_{RAS}).

The application software should ensure that all applicable clock timing specifications are met before the transfer to SDRAM address space which causes the controller to exit self-refresh mode. If a transfer occurs to SDRAM address space when the SCTLE bit is cleared, an internal bus error is generated, and the access does not occur externally, leaving the SDRAM in self-refresh mode. For more information, see "Error Detection" on page 7-7.

No Operation Command

The no operation (NOP) command to the SDRAM has no effect on operations currently in progress. The command inhibit command is the same as a NOP command; however, the SDRAM is not chip-selected. When the SDC is actively accessing the SDRAM to insert additional wait states, the NOP command is given. When the SDC is not accessing the SDRAM, the command inhibit command is given (SMS = 1).

SDC SA10 Pin

The SDRAM's A[10] pin follows the truth table below:

- During the precharge command, it is used to indicate a precharge all
- During a bank activate command, it outputs the row address bit
- During read and write commands, it is used to disable auto-precharge

Therefore, the SDC uses a separate SA10 pin with these rules.

Connect the SA10 pin with the SDRAM A[10] pin. Because the ADSP-BF52x processor uses byte addressing, it starts with A[1]. The A[11] pin is left unconnected for SDRAM accesses and is replaced by the SA10 pin.

SDC Programming Model

The following sections provide programming model information for the SDC.

SDC Configuration

After a processor's hardware or software reset, the SDC clocks are enabled; however, the SDC must be configured and initialized. Before programming the SDC and executing the powerup sequence, these steps are required:

1. Ensure the clock to the SDRAM is stable after the power has stabilized for the proper amount of time (typically 100 ms).
2. Write to the SDRAM refresh rate control register (EBIU_SDRRC).

3. Write to the SDRAM memory bank control register (EBIU_SDBCTL).
4. Write to the SDRAM memory global control register (EBIU_SDGCTL).
5. Perform SDRAM access.

The SDRS bit of the SDRAM control status register can be checked to determine the current state of the SDC. If this bit is set, the SDRAM powerup sequence has not been initiated.

The RDIV field of the EBIU_SDRRC register should be written to set the SDRAM refresh rate.

The EBIU_SDBCTL register should be written to describe the sizes /configuration of SDRAM memory (EBSZ and EBCAW) and to enable the external bank (EBE). Prior to the start of the SDRAM powerup sequence, any access to SDRAM address space, regardless of the state of the EBE bit, generates an internal bus error, and the access does not occur externally. For more information, see “Error Detection” on page 7-7.

The powerup latency can be estimated as:

$t_{RP} + (8 \times t_{RFC}) + t_{MRD} + t_{RCD}$

If the external bank remains disabled after the SDRAM powerup sequence has completed, any transfers to it will result in a hardware error interrupt and the SDRAM transfer will not occur.

The EBIU_SDGCTL register is written:

- To set the SDRAM cycle timing options (CL, TRAS, TRP, TRCD, TWR, EBUFE)
- To enable the SDRAM clock (SCTLE)
- To select and enable the start of the SDRAM powerup sequence (PSM, PSSE)

If SCTLE is disabled, any access to SDRAM address space generates an internal bus error and the access does not occur externally. For more information, see “Error Detection” on page 7-7.

Once the PSSE bit in the EBIU_SDGCTL register is set, and a transfer occurs to enabled SDRAM address space, the SDC initiates the SDRAM powerup sequence. The exact sequence is determined by the PSM bit in the EBIU_SDGCTL register. The transfer used to trigger the SDRAM powerup sequence can be either a read or a write. This transfer occurs when the SDRAM powerup sequence has completed. This initial transfer takes many cycles to complete since the SDRAM powerup sequence must take place.

Example SDRAM System Block Diagrams

Figure 7-12 shows a block diagram of an SDRAM interface. In this example, the SDC is connected to 2 x (8M x 8) = 8M x 16 to form one external

128M bit / 16M byte bank of memory. The system's page size is 1024 bytes. The same address and control bus feeds both SDRAM devices.

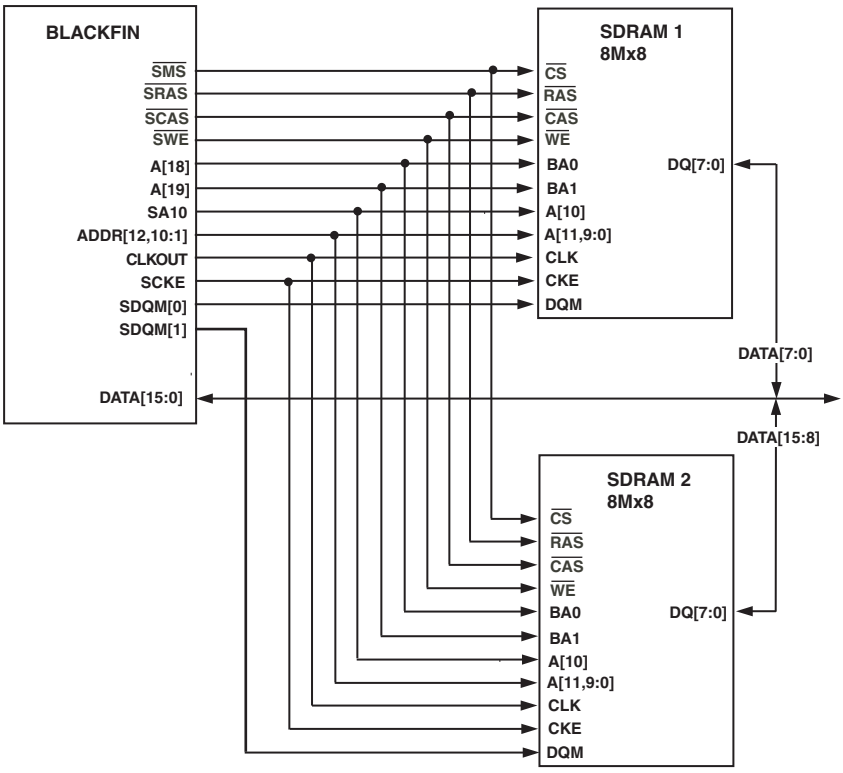


Figure 7-12. SDRAM System Block Diagram, Example 1

Figure 7-13 shows a block diagram of an SDRAM interface. In this example, the SDC is connected to 4 x (16M x 4) = 16M x 16 to form one external 256M bit / 32M byte bank of memory. The system's page size is

2048 bytes. The same address and control bus pass a registered buffer before they feed all 4 SDRAM devices.

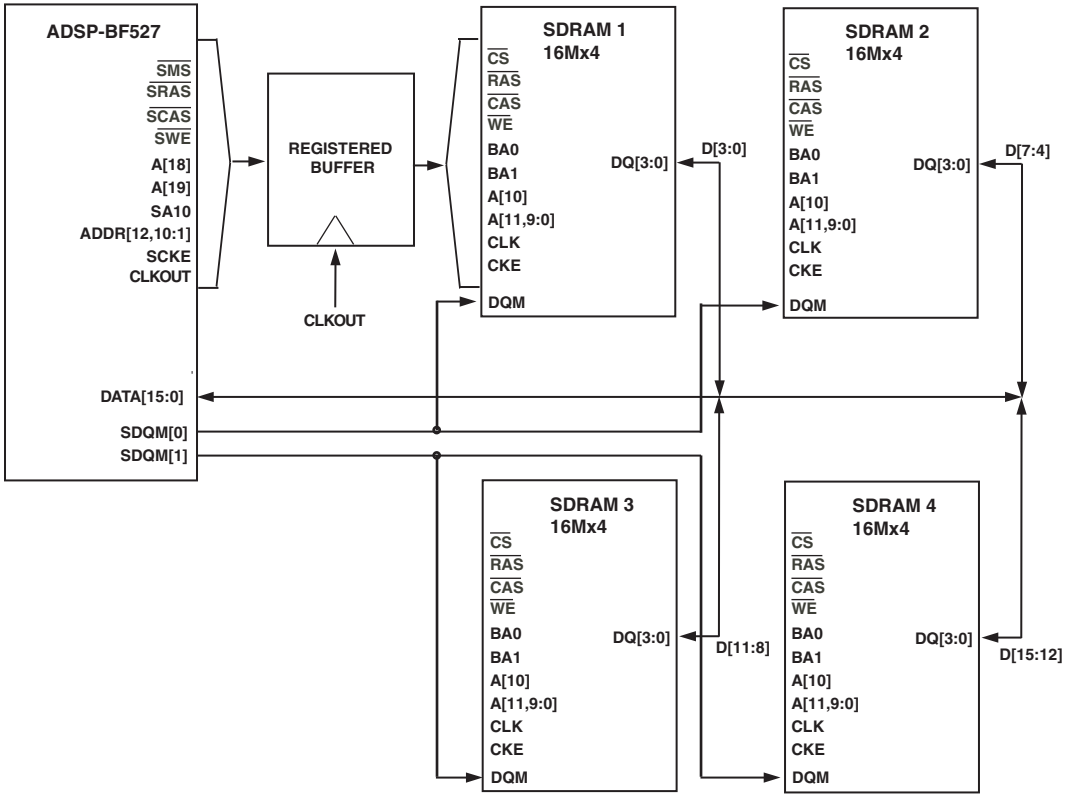


Figure 7-13. SDRAM System Block Diagram, Example 2

Furthermore, the EBUFE bit should be used to enable or disable external buffer timing. When buffered SDRAM modules or discrete register-buffers are used to drive the SDRAM control inputs, EBUFE should be set. Using this setting adds a cycle of data buffering to read and write accesses.

SDC Register Definitions

The following sections describe the SDC registers.

EBIU_SDRRC Register

The SDRAM refresh rate control register (EBIU_SDRRC, shown in Figure 7-14) provides a flexible mechanism for specifying the auto-refresh timing. Since the clock supplied to the SDRAM can vary, the SDC provides a programmable refresh counter, which has a period based on the value programmed into the RDIV field of this register. This counter coordinates the supplied clock rate with the SDRAM device's required refresh rate.

The desired delay (in number of SDRAM clock cycles) between consecutive refresh counter time-outs must be written to the RDIV field. A refresh counter time-out triggers an auto-refresh command to all external SDRAM devices. Write the RDIV value to the EBIU_SDRRC register before the SDRAM powerup sequence is triggered. Change this value only when the SDC is idle.

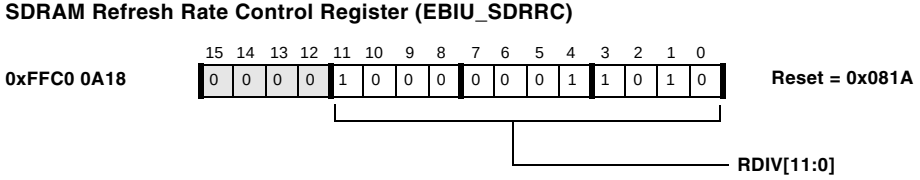

SDRAM Refresh Rate Control Register (EBIU_SDRRC)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0A18
0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 0
Reset = 0x081A
RDIV[11:0]

Figure 7-14. SDRAM Refresh Rate Control Register

To calculate the value that should be written to the EBIU_SDRRC register, use the following equation:

$$RDIV = ((f_{SCLK} \times t_{REF}) / NRA) - (t_{RAS} + t_{RP})$$

$$= (f_{SCLK} \times t_{REFI}) - (t_{RAS} + t_{RP})$$

Where:

- f_{SCLK} = SDRAM clock frequency (system clock frequency)
- t_{REF} = SDRAM row refresh period
- t_{REFI} = SDRAM row refresh interval
- NRA = Number of row addresses in SDRAM (refresh cycles to refresh whole SDRAM)
- t_{RAS} = Active to precharge time (`TRAS` in the SDRAM memory global control register) in number of clock cycles
- t_{RP} = RAS to precharge time (`TRP` in the SDRAM memory global control register) in number of clock cycles

This equation calculates the number of clock cycles between required refreshes and subtracts the required delay between bank activate commands to the same internal bank ($t_{RC} = t_{RAS} + t_{RP}$). The t_{RC} value is subtracted, so that in the case where a refresh time-out occurs while an SDRAM cycle is active, the SDRAM refresh rate specification is guaranteed to be met. The result from the equation should always be rounded down to an integer.

Below is an example of the calculation of `RDIV` for a typical SDRAM in a system with a 133 MHz clock:

- f_{SCLK} = 133 MHz
- t_{REF} = 64 ms
- NRA = 8192 row addresses
- t_{RAS} = 6
- t_{RP} = 3

The equation for RDIV yields:
RDIV = ((133 x 10^6 x 64 x 10^{-3}) / 8192) – (6 + 3) = 1030 clock cycles

This means RDIV is 0x406 and the EBIU_SDRRC register should be written with 0x406.

RDIV must be programmed to a nonzero value if the SDRAM controller is enabled. When RDIV = 0, operation of the SDRAM controller is not supported and can produce undesirable behavior. Values for RDIV can range from 0x001 to 0xFFF.

EBIU_SDBCTL Register

The SDRAM memory bank control register (EBIU_SDBCTL), shown in Figure 7-15, includes external bank-specific programmable parameters. It allows software to control some parameters of the SDRAM. The external bank can be configured for a different size of SDRAM. It uses the access timing parameters defined in the SDRAM memory global control register (EBIU_SDGCTL). The EBIU_SDBCTL register should be programmed before powerup and should be changed only when the SDC is idle.

- External bank enable (EBE)

  The EBE bit is used to enable or disable the external SDRAM bank. If the SDRAM is disabled, any access to the SDRAM address space generates an internal bus error, and the access does not occur externally. For more information, see “Error Detection” on page 7-7.

- External bank size (EBSZ)

  The EBSZ encoding stores the configuration information for the SDRAM bank interface. The EBIU supports 64M bit, 128M bit, 256M bit, and 512M bit SDRAM devices with x4, x8, and x16

configurations. Table 7-13 maps SDRAM density and I/O width. See "SDRAM External Bank Size" on page 7-26 for more information regarding the decoding of bank start addresses.

- External bank column address width (EBCAW)

  The SDC determines the internal SDRAM page size from the EBCAW parameters. Page sizes of 512 B, 1K byte, 2K byte, and 4K byte are supported. Table 7-13 shows the page size and breakdown of the internal address (IA[31:0], as seen from the core or DMA) into the row, bank, column, and byte address. The bank width in all cases is 16 bits. The column address and the byte address together make up the address inside the page.

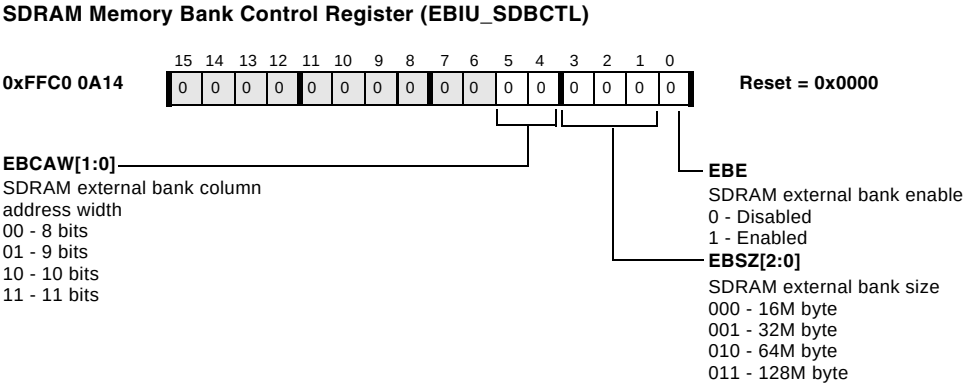

SDRAM Memory Bank Control Register (EBIU_SDBCTL)
0xFFC0 0A14
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
EBCAW[1:0]
SDRAM external bank column address width
00 - 8 bits
01 - 9 bits
10 - 10 bits
11 - 11 bits
EBE
SDRAM external bank enable
0 - Disabled
1 - Enabled
EBSZ[2:0]
SDRAM external bank size
000 - 16M byte
001 - 32M byte
010 - 64M byte
011 - 128M byte

Figure 7-15. SDRAM Memory Bank Control Register

The page size can be calculated for 16-bit SDRAM banks with this formula:

page size = $2^{(CAW + 1)}$

where CAW is the column address width of the SDRAM, plus 1 because the SDRAM bank is 16 bits wide (1 address bit = 2 bytes).

Table 7-13. Internal Address Mapping

Bank Size (M byte) EBSZ bits	Col. Addr. Width (CAW) EBCAW bits	Page Size (K Byte)	Bank Address	Row Address	Page	
					Column Address	Byte Address
128	11	4	IA[26:25]	IA[24:12]	IA[11:1]	IA[0]
128	10	2	IA[26:25]	IA[24:11]	IA[10:1]	IA[0]
128	9	1	1A[26:25]	IA[24:10]	IA[9:1]	IA[0]
128	8	0.5	IA[26:25]	IA[24:9]	IA[8:1]	IA[0]
64	11	4	IA[25:24]	IA[23:12]	IA[11:1]	IA[0]
64	10	2	IA[25:24]	IA[23:11]	IA[10:1]	IA[0]
64	9	1	IA[25:24]	IA[23:10]	IA[9:1]	IA[0]
64	8	0.5	IA[25:24]	IA[23:9]	IA[8:1]	IA[0]
32	11	4	IA[24:23]	IA[22:12]	IA[11:1]	IA[0]
32	10	2	IA[24:23]	IA[22:11]	IA[10:1]	IA[0]
32	9	1	IA[24:23]	IA[22:10]	IA[9:1]	IA[0]
32	8	0.5	IA[24:23]	IA[22:9]	IA[8:1]	IA[0]
16	11	4	IA[23:22]	IA[21:12]	IA[11:1]	IA[0]
16	10	2	IA[23:22]	IA[21:11]	IA[10:1]	IA[0]
16	9	1	IA[23:22]	IA[21:10]	IA[9:1]	IA[0]
16	8	0.5	IA[23:22]	IA[21:9]	IA[8:1]	IA[0]

Using SDRAMs With Systems Smaller than 16M byte

It is possible to use SDRAMs smaller than 16M byte on the ADSP-BF52x, as long as it is understood how the resulting memory map is altered. Figure 7-16 shows an example where a 2M byte SDRAM (512K x 16 bits x 2 banks) is mapped to the external memory interface. In this example,

there are 11 row addresses and eight column addresses per bank. Referring to Table 7-4 on page 7-25, the lowest available bank size (16M byte) for a device with eight column addresses has two bank address lines (IA[23:22]) and 13 row address lines (IA[21:9]). Therefore, one processor bank address line and two row address lines are unused when hooking up to the SDRAM in the example. This causes aliasing in the processor's external memory map, which results in the SDRAM being mapped into non-contiguous regions of the processor's memory space.

Referring to the table in Figure 7-16, note that each line in the table corresponds to 2^{19} bytes, or 512K byte. Thus, the mapping of the 2M byte SDRAM is non-contiguous in Blackfin memory, as shown by the memory mapping in the left side of the figure.

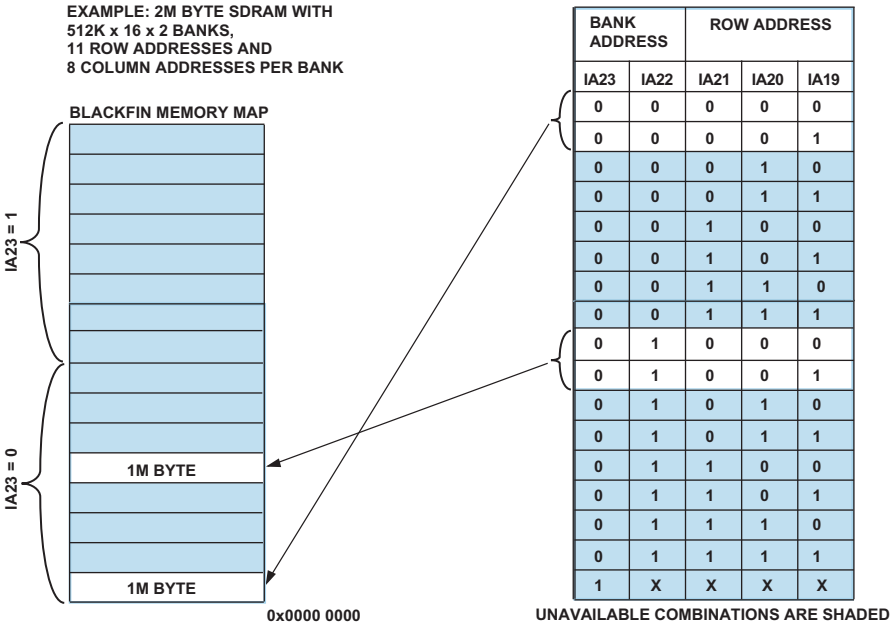

EXAMPLE: 2M BYTE SDRAM WITH
512K x 16 x 2 BANKS,
11 ROW ADDRESSES AND
8 COLUMN ADDRESSES PER BANK
BLACKFIN MEMORY MAP
IA23 = 1
IA23 = 0
1M BYTE
1M BYTE
0x0000 0000
BANK ADDRESS
ROW ADDRESS
IA23 IA22 IA21 IA20 IA19
0 0 0 0 0
0 0 0 0 1
0 0 0 1 0
0 0 0 1 1
0 0 1 0 0
0 0 1 0 1
0 0 1 1 0
0 0 1 1 1
0 1 0 0 0
0 1 0 0 1
0 1 0 1 0
0 1 0 1 1
0 1 1 0 0
0 1 1 0 1
0 1 1 1 0
0 1 1 1 1
1 X X X X
UNAVAILABLE COMBINATIONS ARE SHADED

Figure 7-16. Using Small SDRAMs

EBIU_SDGCTL Register

The SDRAM memory global control register (EBIU_SDGCTL) includes all programmable parameters associated with the SDRAM access timing and configuration. Figure 7-17 shows the EBIU_SDGCTL register bit definitions.

SDRAM clock enable (SCTLE)

The SCTLE bit is used to enable or disable the SDC. If SCTLE is cleared, any access to SDRAM address space generates an internal bus error, and the access does not occur externally. For more information, see "Error Detection" on page 7-7. When SCTLE is cleared, all SDC control pins are in their inactive states and the SDRAM clock is not running. The SCTLE bit must be set for SDC operation and is set by default at reset. The CAS latency (CL), SDRAM t_{RAS} timing (TRAS), SDRAM t_{RP} timing (TRP), SDRAM t_{RCD} timing (TRCD), and SDRAM t_{WR} timing (TWR) bits should be programmed based on the system clock frequency and the timing specifications of the SDRAM used.

The SCTLE bit allows software to disable all SDRAM control pins. These pins are SDQM[3:0], $\overline{\text{SCAS}}$, $\overline{\text{SRAS}}$, $\overline{\text{SWE}}$, SCKE, and CLKOUT.

- SCTLE = 0
  Disable all SDRAM control pins (control pins negated, CLKOUT low).
- SCTLE = 1
  Enable all SDRAM control pins (CLKOUT toggles).

Note that the CLKOUT function is also shared with the AMC. Even if SCTLE is disabled, CLKOUT can be enabled independently by the CLKOUT enable in the AMC (AMCKEN in the EBIU_AMGCTL register).

If the system does not use SDRAM, SCTLE should be set to 0.

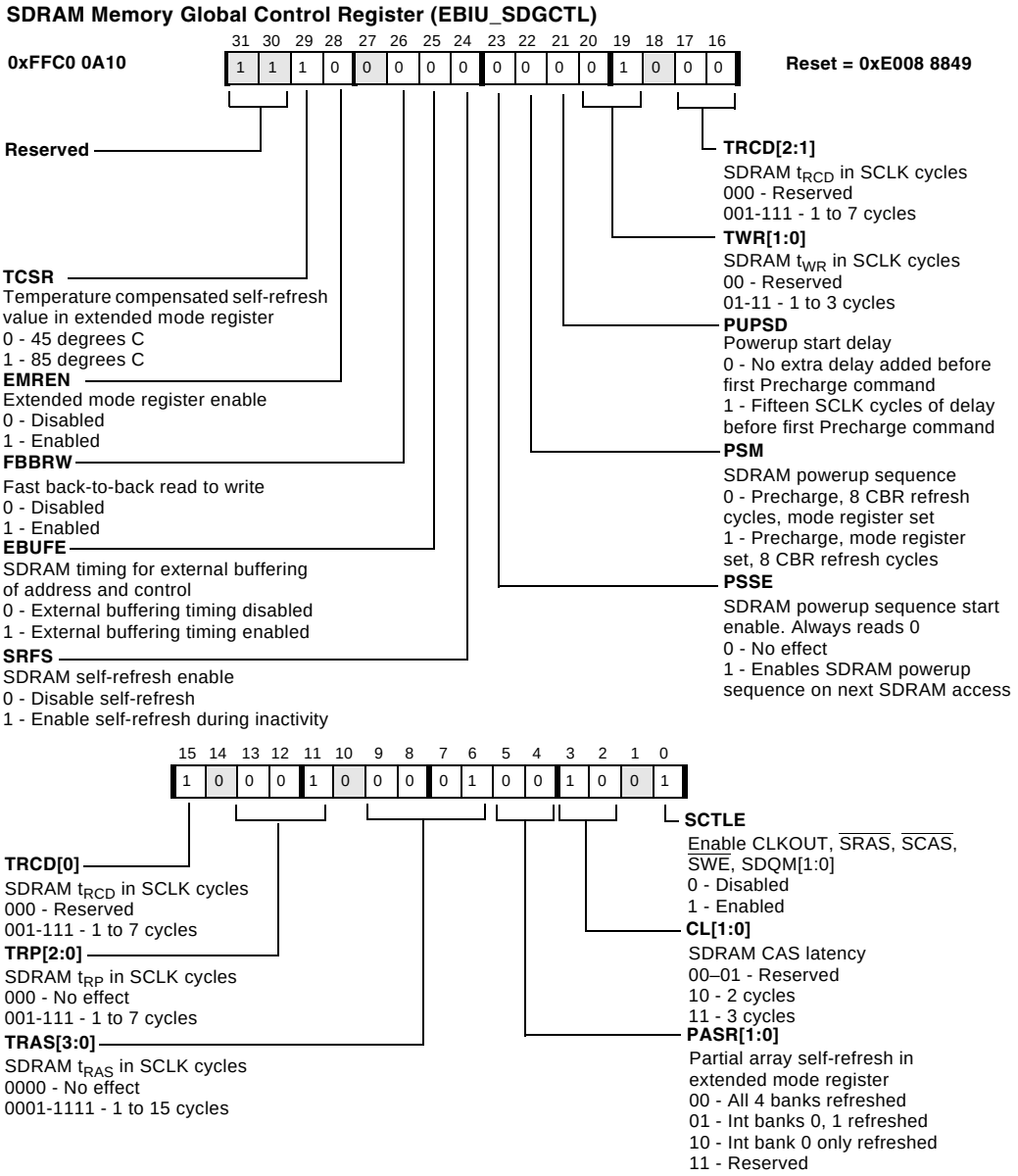

SDRAM Memory Global Control Register (EBIU_SDGCTL)
0xFFC0 0A10
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
1 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0
Reset = 0xE008 8849
Reserved
TRCD[2:1]
SDRAM tRCD in SCLK cycles
000 - Reserved
001-111 - 1 to 7 cycles
TWR[1:0]
SDRAM tWR in SCLK cycles
00 - Reserved
01-11 - 1 to 3 cycles
PUPSD
Powerup start delay
0 - No extra delay added before first Precharge command
1 - Fifteen SCLK cycles of delay before first Precharge command
PSM
SDRAM powerup sequence
0 - Precharge, 8 CBR refresh cycles, mode register set
1 - Precharge, mode register set, 8 CBR refresh cycles
PSSE
SDRAM powerup sequence start enable. Always reads 0
0 - No effect
1 - Enables SDRAM powerup sequence on next SDRAM access
TCSR
Temperature compensated self-refresh value in extended mode register
0 - 45 degrees C
1 - 85 degrees C
EMREN
Extended mode register enable
0 - Disabled
1 - Enabled
FBBRW
Fast back-to-back read to write
0 - Disabled
1 - Enabled
EBUFE
SDRAM timing for external buffering of address and control
0 - External buffering timing disabled
1 - External buffering timing enabled
SRFS
SDRAM self-refresh enable
0 - Disable self-refresh
1 - Enable self-refresh during inactivity
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 0 0 1 0 0 0 0 1 0 0 1 0 0 1
SCTLE
Enable CLKOUT, SRAS, SCAS, SWE, SDQM[1:0]
0 - Disabled
1 - Enabled
CL[1:0]
SDRAM CAS latency
00–01 - Reserved
10 - 2 cycles
11 - 3 cycles
PASR[1:0]
Partial array self-refresh in extended mode register
00 - All 4 banks refreshed
01 - Int banks 0, 1 refreshed
10 - Int bank 0 only refreshed
11 - Reserved
TRCD[0]
SDRAM tRCD in SCLK cycles
000 - Reserved
001-111 - 1 to 7 cycles
TRP[2:0]
SDRAM tRP in SCLK cycles
000 - No effect
001-111 - 1 to 7 cycles
TRAS[3:0]
SDRAM tRAS in SCLK cycles
0000 - No effect
0001-1111 - 1 to 15 cycles

Figure 7-17. SDRAM Memory Global Control Register

If an access occurs to the SDRAM address space while SCTLE is 0, the access generates an internal bus error and the access does not occur externally. For more information, see “Error Detection” on page 7-7.

With careful software control, the SCTLE bit can be used in conjunction with the SRFS bit to further lower power consumption by freezing the CLKOUT pin. However, SCTLE must remain enabled at all times when the SDC is needed to generate auto-refresh commands to SDRAM.

CAS latency (CL)

The CL bits in the SDRAM memory global control register (EBIU_SDGCTL) select the CAS latency value:

CL = b#00 reserved

CL = b#01 reserved

CL = b#10 two clock cycles

CL = b#11 three clock cycles

Partial array self refresh (PASR)

The PASR bits determine how many internal SDRAM banks are refreshed during self-refresh.

PASR = b#00 all 4 banks

PASR = b#01 internal banks 0 and 1 refreshed

PASR = b#10 only internal bank 0 refreshed

PASR = b#11 reserved

Internal banks are decoded with the A[19:18] pins.

The PASR feature requires careful software control with regard to the internal bank used.

Bank activate command delay (TRAS)

The TRAS bits in the SDRAM memory global control register (EBIU_SDGCTL) select the t_{RAS} value. Any value between 1 and 15 clock cycles can be selected. For example:

TRAS = b#0000 no effect

TRAS = b#0001 one clock cycle

TRAS = b#0010 two clock cycles

TRAS = b#1111 fifteen clock cycles

Bank precharge delay (TRP)

The TRP bits in the SDRAM memory global control register (EBIU_SDGCTL) select the t_{RP} value. Any value between 1 and 7 clock cycles may be selected. For example:

TRP = b#000 no effect

TRP = b#001 one clock cycle

TRP = b#010 two clock cycles

TRP = b#111 seven clock cycles

RAS to CAS delay (TRCD)

The TRCD bits in the SDRAM memory global control register (EBIU_SDGCTL) select the t_{RCD} value Any value between 1 and 7 clock cycles may be selected. For example:

TRCD = b#000 reserved, no effect

TRCD = b#001 one clock cycle

TRCD = b#010 two clock cycles

TRCD = b#111 seven clock cycles

Write to precharge delay (TWR)

The TWR bits in the SDRAM memory global control register (EBIU_SDGCTL) select the t_{WR} value. Any value between 1 and 3 clock cycles may be selected. For example:

TWR = b#00 reserved

TWR = b#01 one clock cycle

TWR = b#10 two clock cycles

TWR = b#11 three clock cycles

Power-Up Start Delay (PUPSD)

The power-up start delay bit (PUPSD) optionally delays the power-up start sequence for 15 SCLK cycles. This is useful for multiprocessor systems sharing an external SDRAM. If the bus has been previously granted to the other processor before power-up and self-refresh mode is used when switching bus ownership, then the PUPSD bit can be used to guarantee a sufficient period of inactivity from self-refresh to the first Precharge command in the power-up sequence in order to meet the exit self-refresh time (t_{XSR}) of the SDRAM.

Power-Up Sequence Mode (PSM)

If the PSM bit is set to 1, the SDC command sequence is:

1. precharge all
2. mode register set
3. eight auto-refresh cycles

If the PSM bit is cleared, the SDC command sequence is:

1. precharge all
2. eight auto-refresh cycles
3. mode register set

Power-Up Sequence Start Enable (PSSE)

The PSM and PSSE bits work together to specify and trigger an SDRAM power-up (initialization) sequence. Two events must occur before the SDC does the SDRAM power-up sequence:

- The PSSE bit must be set to enable the SDRAM power-up sequence.
- A read or write access must be done to enabled SDRAM address space in order to have the external bus granted to the SDC so that the SDRAM power-up sequence may occur.

The SDRAM power-up sequence occurs and is followed immediately by the read or write transfer to SDRAM that was used to trigger the SDRAM power-up sequence. Note that there is a latency for this first access to

SDRAM because the SDRAM power-up sequence takes many cycles to complete.

Before executing the SDC power-up sequence, ensure that the SDRAM receives stable power and is clocked for the proper amount of time, as described in the SDRAM specifications.

Self-Refresh Setting (SRFS)

The SRFS and SCTLE bits work together in EBIU_SDGCTL for self-refresh control.

SRFS = b#0 disable self-refresh mode

SRFS = b#1 enter self-refresh mode

When SRFS is set, self-refresh mode is triggered. Once the SDC completes any active transfers, the SDC executes a sequence of commands to put the SDRAM into self-refresh mode.

When the device comes out of reset, the SCKE pin is driven high. If it is necessary to enter self-refresh mode after reset, program SRFS = b#1.

Enter Self-Refresh Mode

When SRFS is set, once the SDC enters an idle state it issues a precharge all command and then issues a self-refresh entry command. If an internal access is pending, the SDC delays issuing the self-refresh entry command until it completes the pending SDRAM access and any subsequent pending access requests.

Once the SDRAM device enters into self-refresh mode, the SDRAM controller asserts the SDSRA bit in the SDRAM control status register (EBIU_SDSTAT).

Once the SRFS bit is set to 1, the SDC enters self-refresh mode when it finishes pending accesses. There is no way to cancel the entry into self-refresh mode.

Before disabling the CLKOUT pin with the SCTLE bit, be sure to place the SDC in self-refresh mode (SRFS bit). If this is not done, the SDRAM is unclocked and will not work properly.

Exit Self-Refresh Mode

The SDRAM device exits self-refresh mode only when the SDC receives core or DMA requests. In conjunction with the SRFS bit, two possibilities are given to exit self-refresh mode.

- If the SRFS bit remains set before the core/DMA request, the SDC exits self-refresh mode temporarily for a single request and returns back to self-refresh mode until a new request is latched.
- If the SRFS bit is cleared before the core/DMA request, the SDC exits self-refresh mode and returns to auto-refresh mode.

Before exiting self-refresh mode with the SRFS bit, be sure to enable the CLKOUT pin (SCTLE bit). If this is not done, the SDRAM is unclocked and will not work properly.

External buffering enabled (EBUFE)

With the total I/O width of 16 bits, a maximum of 4x4 bits can be connected in parallel in order to increase the system's overall page size.

To meet overall system timing requirements, systems that employ several SDRAM devices connected in parallel may require buffering between the

processor and the multiple SDRAM devices. This buffering generally consists of a register and driver.

To meet such timing requirements and to allow intermediary registration, the SDC supports pipelining of SDRAM address and control signals.

The EBUFE bit in the EBIU_SDGCTL register enables this mode:

EBUFE = 0 disable external buffering timing

EBUFE = 1 enable external buffering timing

When EBUFE = 1, the SDRAM controller delays the data in write accesses by one cycle, enabling external buffer registers to latch the address and controls. In read accesses, the SDRAM controller samples data one cycle later to account for the one-cycle delay added by the external buffer registers. When external buffering timing is enabled, the latency of all accesses is increased by one cycle.

Connection of 4 x 4 bits rather than 1 x 16 bits increases the page size by a factor of four, thus resulting in fewer off-page penalties.

Fast Back-to-Back Read to Write (FBBRW)

The FBBRW bit enables an SDRAM read followed by write to occur on consecutive cycles. In many systems, this is not possible because the turn-off time of the SDRAM data pins is too long, leading to bus contention with the succeeding write from the processor. When this bit is cleared, a clock cycle is inserted between read accesses followed immediately by write accesses.

Extended Mode Register Enabled (EMREN)

The EMREN bit enables programming of the extended mode register during startup. The extended mode register is used to control SDRAM power consumption in certain mobile low power SDRAMs. If the EMREN bit is

enabled, then the TCSR and PASR[1:0] bits control the value written to the extended mode register.

Temperature Compensated Self-Refresh (TCSR)

The TCSR bit signals to the SDRAM the worst case temperature range for the system, and thus how often the SDRAM internal banks need to be refreshed during self-refresh.

All reserved bits in this register must always be written with 0s.

EBIU_SDSTAT Register

The SDRAM control status register (EBIU_SDSTAT), shown in Figure 7-18, provides information on the state of the SDC. This information can be

used to determine when it is safe to alter SDC control parameters or it can be used as a debug aid.

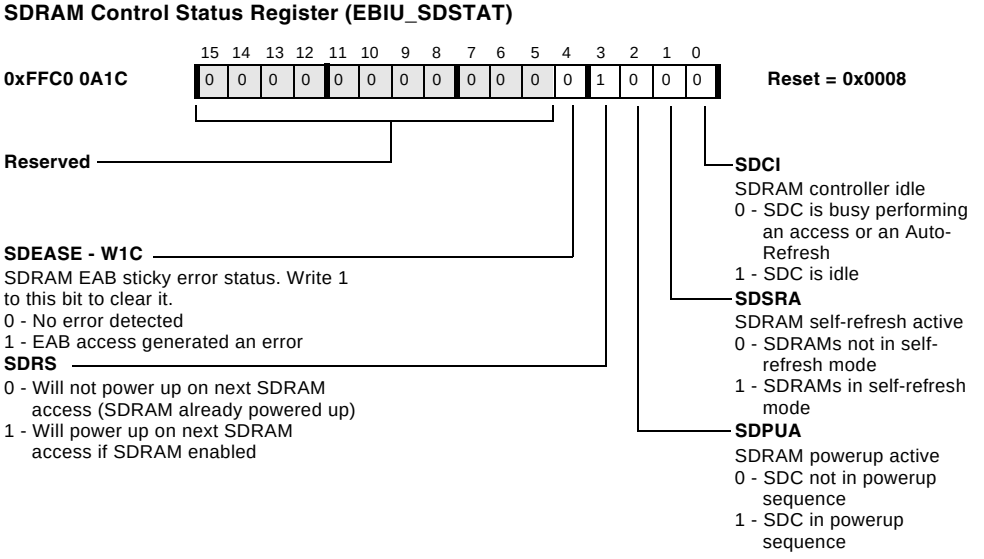

SDRAM Control Status Register (EBIU_SDSTAT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0A1C
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
Reset = 0x0008
Reserved
SDEASE - W1C
SDRAM EAB sticky error status. Write 1 to this bit to clear it.
0 - No error detected
1 - EAB access generated an error
SDRS
0 - Will not power up on next SDRAM access (SDRAM already powered up)
1 - Will power up on next SDRAM access if SDRAM enabled
SDCI
SDRAM controller idle
0 - SDC is busy performing an access or an Auto-Refresh
1 - SDC is idle
SDSRA
SDRAM self-refresh active
0 - SDRAMs not in self-refresh mode
1 - SDRAMs in self-refresh mode
SDPUA
SDRAM powerup active
0 - SDC not in powerup sequence
1 - SDC in powerup sequence

Figure 7-18. SDRAM Control Status Register

- SDC idle (SDCI)

  If the SDCI bit is cleared, the SDC is performing a user access or auto-refresh. If the SDCI bit is set, no commands are issued and the SDC is in idle state.

- SDC self-refresh active (SDSRA)

  If the SDSRA bit is cleared, the SDC is performing auto-refresh (SCKE pin = 0). If the SDSRA bit is set, the SDC performs self-refresh mode (SCKE pin = 1).

- SDC powerup active (SDPUA)

If the SDPUA bit is cleared, the SDC is not in powerup sequence. If the SDPUA bit is set, the SDC performs the powerup sequence.

- SDC powerup delay (SDRS)

  If the SDRS bit is cleared, the SDC has already powered up. If the SDRS bit is set, the SDC will still perform the powerup sequence.

- SDC EAB sticky error status (SDEASE)

  If the SDEASE bit is cleared, there were no errors detected on the EAB core bus. If the SDEASE bit is set, there were errors detected on the EAB core bus. The SDEASE bit is sticky. Once it has been set, software must explicitly write a 1 to the bit to clear it. Writes have no effect on the other status bits, which are updated by the SDC only.

SDC Programming Examples

Listing 7-3 through Listing 7-6 provide examples for working with the SDC.

Listing 7-3. 16-Bit Core Transfers to SDRAM

```
.section L1_data_b;
.byte2 source[N] = 0x1122, 0x3344, 0x5566, 0x7788;
.section SDRAM;
.byte2 dest[N];
.section L1_code;
  I0.L = lo(source);
  I0.H = hi(source);
  I1.L = lo(dest);
  I1.H = hi(dest);
      R0.L = w[I0++];
      p5=N-1;
```

```
      lsetup(lp, lp) lc0=p5;
lp:R0.L = w[I0++] || w[I1++] = R0.L;
                             w[I1++] = R0.L;
```

Listing 7-4. 8-Bit Core Transfers to SDRAM Using Byte Mask SDQM[1:0] Pins

```
.section L1_data_b;
.byte source[N] = 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88;

.section SDRAM;
.byte dest[N];

  p0.L = lo(source);
  p0.H = hi(source);
  p1.L = lo(dest);
  p1.H = hi(dest);
    p5=N;
    lsetup(start, end) lc0=p5;
start:  R0 = b[p0++](z);
end:    b[p1++] = R0;  /* byte data masking */
```

Listing 7-5. Self-refresh Mode Power Savings with Disabled CLKOUT

```
    r0.l = w[I1++];    /* SDRAM access */
  ssync;    /* force last SDRAM access to finish */
  P0.L = lo(EBIU_SDGCTL);
  P0.H = hi(EBIU_SDGCTL);
  R1 = [P0];
  bitset(R1, bitpos(SRFS));    /* enter self-refresh */
  [P0] = R1;
  ssync;

  P0.L = lo(EBIU_SDSTAT);
```

```
  P0.H = hi(EBIU_SDSTAT);
self_refresh_status:
  R0 = [P0];
  ssync;
  cc = bittst(R0, bitpos(SDSRA));  /* poll self-refresh status */
  if !cc jump self_refresh_status;

  P0.L = lo(EBIU_SDGCTL);
  P0.H = hi(EBIU_SDGCTL);
  R1 = [P0];
  bitclr(R1, bitpos(SCTLE));
                   /* disable CLKOUT after approx 20 cycles */
  [P0] = R1;
  ssync;

         P5 = 30000;
         LSETUP(lp,lp) LC0 = P5;
    lp:  nop;  /* dummy loop */

  R1 = [P0];
  bitset(R1, bitpos(SCTLE));
                   /* enable CLKOUT after approx 20 cycles */
  [P0] = R1;
  ssync;

  R1 = [P0];
  bitclr(R1, bitpos(SRFS));   /* exit self-refresh */
  [P0] = R1;
  ssync;

    w[I1++] = r0.l;   /* SDRAM access */
```

Listing 7-6. Init

```
/*********************************************************/
/* SDRAM part# Micron MT48LC32M8A2-75 (32Mx8/256Mbit) */
/* 8k rows, 1k columns          -> EBCAW = 10 */
/* 2xSDRAM: 32Mx16 = 64Mbytes   -> EBSZ = 010 */

/* populated SDRAM addresses    -> 0x00000000 - 0x01FFFFFF */
/* internal SDRAM bank A  0x00000000 - 0x007FFFFF */
/* internal SDRAM bank B  0x00800000 - 0x00FFFFFF */
/* internal SDRAM bank C  0x01000000 - 0x017FFFFF */
/* internal SDRAM bank D  0x01800000 - 0x01FFFFFF */

/* powerup: PRE-REF-MRS     -> PSM = 0 */
/* SCLK = 133 MHz */
/* tCK = 7.5ns min@CL=3    -> CL = 3 */
/* tRAS = 44ns min          -> TRAS = 6 */
/* tRP = 20ns min           -> TRP = 3 */
/* tRCD = 20ns min          -> TRCD = 3 */
/* tWR = 15ns min           -> TWR = 2 */
/* tREF = 64ms max
                ->RDIV = (133MHz*64ms)/8192-(6+3)=0x406 cycles */
/*********************************************************/
#ifdef INIT_SDRAM

  /* Check if already enabled */
  p0.l = lo(EBIU_SDSTAT);
  p0.h = hi(EBIU_SDSTAT);
  r0 = [p0];
  cc = bittst(r0, bitpos(SDRS));
  if !cc jump skip init_sdram;

  /* SDRAM Refresh Rate Control Register */
```

```
  P0.L = lo(EBIU_SDRRC);
  P0.H = hi(EBIU_SDRRC);
  R0.L = 0x0406;
  W[P0] = R0.L;

  /* SDRAM Memory Bank Control Register */
  P0.L = lo(EBIU_SDBCTL);
  P0.H = hi(EBIU_SDBCTL);
  R0.L = 0x0025;
  W[P0] = R0.L;

/* SDRAM Memory Global Control Register */
  P0.L = lo(EBIU_SDGCTL);
  P0.H = hi(EBIU_SDGCTL);
  R0.L = 0x998d;
  R0.H = 0x8491;
  [P0] = R0;
  ssync;   /* wait until executed */
```

8 HOST DMA PORT

This chapter describes the Host DMA Port (HOSTDP). Following an overview and a list of key features are a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Overview

The HOSTDP facilitates a host device external to the ADSP-BF52x processor to be a Direct Memory Access (DMA) master and transfer data back and forth. The host device always masters the transactions and the Blackfin processor is always a DMA slave device.

It is necessary to pay particular attention to nomenclature involving the Host DMA Port. The Host DMA Port is sometimes abbreviated as HOSTDP. Register and pin names all have the prefix `HOST_`. The HOSTDP is a peripheral on the ADSP-BF52x processor, which is referred to as the slave processor or Blackfin slave. The host processor is also referred to as the host, master, external host, or external master.

The HOSTDP is enabled through the Peripheral Access Bus (PAB) interface. Once enabled, the DMA is controlled by an external host. The external host can then program the DMA to send/receive data to any valid internal or external memory location.

Features

The HOSTDP controller includes the following features:

- Allows external master to configure DMA READ/WRITE data transfers and read port status
- Uses a flexible asynchronous memory protocol for external interface
- 8/16-bit external data interface to host device
- Half-duplex operation
- Little/Big Endian data transfer
- Internal FIFO which holds sixteen 16-bit words
- Acknowledge Mode allows flow control on host transactions
- Interrupt Mode guarantees a burst of FIFO depth host transactions
- Ability to enable and disable data reads/writes
- DMA bandwidth control

Interface Overview

Table 8-1 defines the pins for the HOSTDP interface. The interface uses a flexible asynchronous memory interface, which can be gluelessly connected to a variety of host processors.

Table 8-1. HOSTDP External Pins

Pin	Description
Port H - HOST_DATA <15:0>	16-bit data port
PG15- $\overline{\text{HOST\_CE}}$	Chip Enable for the HOSTDP
PG11 - $\overline{\text{HOST\_WR}}$	Write strobe
PG14- $\overline{\text{HOST\_RD}}$	Read strobe
PH13 - HOST_ADDR	Address pin 0: data port access 1: configuration port access
PH12 - HOST_ACK (HRDY/FRDY)	Flow control pin: HRDY-Acknowledge mode & FRDY- Interrupt mode

Due to the Blackfin processor's use of multiplexed pins, utilizing the Host DMA Port can preclude the use of other peripherals. The Ethernet MAC and NAND flash are unavailable when using the HOSTDP. Refer to "General-Purpose Ports" on page 9-1 for a complete description of the pin multiplexing scheme.

Description of Operation

The following sections describe the operation of the HOSTDP interface.

Architecture

The HOSTDP block diagram, shown in Figure 8-1, illustrates the overall architecture of the HOSTDP.

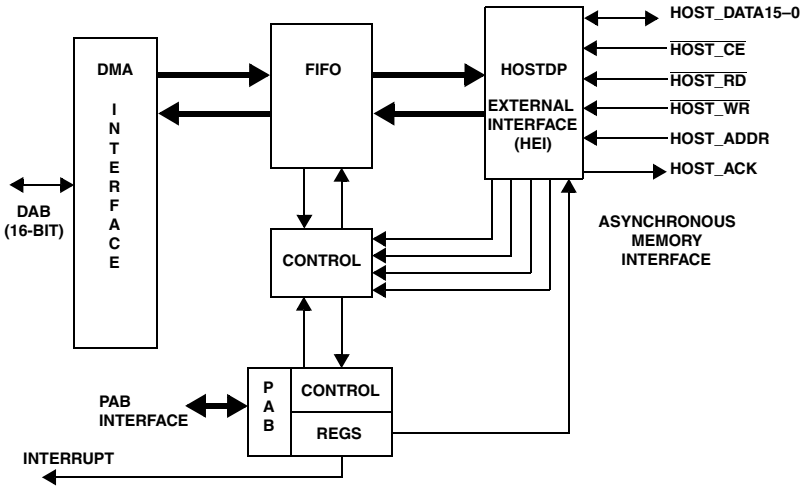


Figure 8-1. HOSTDP Block Diagram

The HOSTDP is enabled/disabled through PAB writes to the HOST_CONTROL register. Once enabled, the HOSTDP interfaces to the external world using asynchronous memory protocol and handshakes with the DMA controller internally using the DMA Access Bus (DAB). The HOSTDP allows the external host to program the DMA to transfer data in either direction. The HOSTDP can be broken into five functional blocks, identified as follows:

- **Host External Interface (HEI)**

  This block interfaces to the external host and, based on inputs from the host device, initiates data or control message transfer.

- **PAB Interface**

The HOSTDP is programmed/queried for status by reads or writes to appropriate registers in this block through the PAB.

- **FIFO**

  A port FIFO is used for data transfers and can store up to sixteen 16-bit words.

- **Control**

  The Control block handles the HOSTDP's different states as well as the handshakes between the external host device and DMA Interfaces.

- **DMA Interface**

  This block is connected to the DAB and interacts with the DMA to transfer control messages and data between DMA and external host device.

Functional Description

The following sections describe the functional operation of the Host DMA Port.

HOSTDP Configuration

Before any data transfer can occur, the DMA engine must be configured by the host processor. Because the host is unaware of the internal state of the Host DMA Port peripheral and its associated DMA activity, the host processor is required to check the `ALLOW_CNFG` bit in `HOST_STATUS` register before attempting configuration writes. Additionally, this status read sets some internal states inside the Host DMA Port. Configuration requires seven 16-bit words to be written in the following order to the configura-

tion port before Host Read Data or Host Write Data operations can occur:

- HOST_CONFIG
- START_ADDR.L
- START_ADDR.H
- XCOUNT
- XMODIFY
- YCOUNT
- YMODIFY

The only word different from the standard DMA described in the DMA chapter is the HOST_CONFIG word, as shown in Figure 8-2.

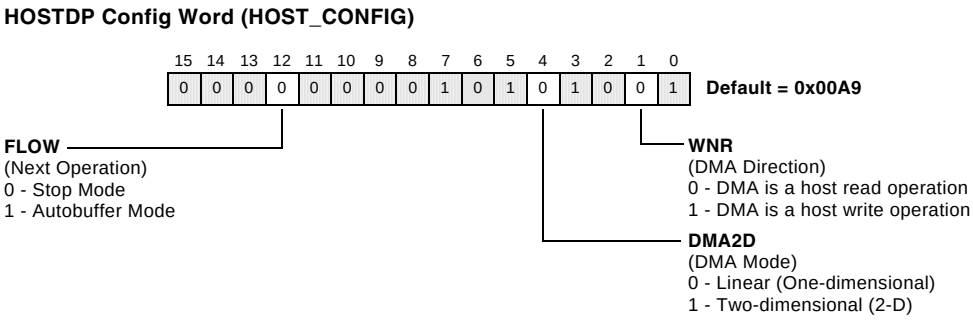

HOSTDP Config Word (HOST_CONFIG)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1
Default = 0x00A9
FLOW
(Next Operation)
0 - Stop Mode
1 - Autobuffer Mode
WNR
(DMA Direction)
0 - DMA is a host read operation
1 - DMA is a host write operation
DMA2D
(DMA Mode)
0 - Linear (One-dimensional)
1 - Two-dimensional (2-D)

Figure 8-2. HOSTDP Configuration Word

Additional information for the HOST_CONFIG bits includes the following:

- **Host DMA Direction** (WNR)

  When this bit is set, the DMA will write to memory (host write). When this bit is cleared, the DMA will read from memory (host read).

- **Host DMA Mode** (DMA2D)

  When this bit is cleared, the DMA is linear one-dimension (1-D). When this bit is set, the DMA is in two-dimension mode (2-D).

- **FLOW** (FLOW)

  When this bit is cleared, the DMA runs in STOP mode. When this bit is set, the DMA runs in AUTOBUFFER mode.

For information on how these words are used to configure the DMA, refer to "Direct Memory Access" on page 6-1.

Before accessing the data port, the host processor must write all seven descriptor words. The HOSTDP module does not forward descriptors to the DMA channel until it has received all seven words. Similarly, the host processor is not permitted to provide new descriptor data before all data words of the former work unit have been transferred. However, the host can truncate an initiated transfer using the DMA_FINISH control command. As always, ALLOW_CNFG in the HOST_STATUS register must be polled before writing a new configuration to the Host DMA Port. Please see "Control Commands Between the External Host and HOSTDP" on page 8-19 for additional information.

Additional latency is incurred when a Host Read Data operation follows a Host Write Data operation. Even though the configuration for the Host Read is complete, the DMA engine must first empty the FIFO for the Host Write operation and then change directions and start filling the FIFO for the Host Read Data operation.

HOSTDP Transactions

The HOSTDP is enabled by writing to the HOSTDP_EN bit of the HOST_CONTROL register. In order to disable the HOSTDP, the HOSTDP_RST bit must be asserted before clearing HOSTDP_EN. There are four types of HOSTDP transactions. Each type of access is based on the HOST_ADDR and whether the $\overline{\text{HOST\_RD}}$ or $\overline{\text{HOST\_WR}}$ signal is asserted.

When chip enable ($\overline{\text{HOST\_CE}}$) is inactive, the HOSTDP stays idle. Modes listed in Table 8-2 are only possible when HOST_CE is active.

Table 8-2. Types of HOSTDP Transactions

Address	$\overline{\text{HOST\_RD}}$	$\overline{\text{HOST\_WR}}$	$\overline{\text{HOST\_CE}}$	Function
0x0	0	1	0	Host Read Data Operation
0x0	1	0	0	Host Write Data Operation
0x1	1	0	0	Host Write Configuration or Control Command
0x1	0	1	0	Host Read HOST_STATUS register

Host Read Status

The Host DMA Port is robust against on-the-fly changes of data direction. However, in acknowledge mode, it is encouraged not to initiate a new work unit with different data direction before the FIFOEMPTY bit in the HOST_STATUS register is cleared. This is to avoid excessive wait states inserted by HRDY.

The external host can read the HOST_STATUS register at any time. By performing this operation, the external host can query the status of the HOSTDP. Note that in 8-bit configurations, the host can only read the lower byte of the HOST_STATUS register. HOST_STATUS can also be read through a PAB access. When accessed through the PAB, all 16 bits of HOST_STATUS are always read. The contents of HOST_STATUS are detailed in "Host DMA Port Registers" on page 8-26.

Host Read Data and Host Write Data Operations

After the HOSTDP has been configured and enabled by way of PAB accesses and the DMA channel has been configured through Host Write Configuration accesses, data can be transferred.

All DMAs between the HOSTDP FIFO and memory are 16-bit transactions. This is important when setting `XMODIFY` and `YMODIFY`. The amount of data moved between the host processor and the HOSTDP must be a multiple of the FIFO depth (sixteen 16-bit words). The user is required to set the `XCOUNT/YCOUNT` values such that this is true and to also ensure that the correct number of host data reads or host data writes are performed.

A Host Write Data operation is used to transfer data from the host to the slave processor. The host performs write transactions and the HOSTDP writes the data from these transactions into its FIFO. The DMA engine concurrently moves data from the HOSTDP's FIFO to the location in memory specified by the DMA configuration words.

A Host Read Data operation is used to transfer data from the slave processor to the host. The DMA engine moves data from the specified location in the Blackfin slave's memory into the HOSTDP's FIFO. The host performs read accesses to read data out of this FIFO.

In the case of host writes, the host processor must "pad" the end of the transfer with dummy data to ensure this (for example, if the host wants 31 words it must send an extra dummy word to equal 32). In the case of host reads, dummy reads must be performed at the end and the host can then throw away the results. This is true in both interrupt mode and acknowledge mode.

When in 8-bit mode, since all DMAs from the HOSTDP are 16 bits, data will be packed into 16-bit words in the HOSTDP FIFO during Host Data Write operations. For Host Data Read operations in 8-bit mode, data will be unpacked in the FIFO into 8-bit words for transmission. Because all DMAs are 16-bit and the data bus is either 8-bit or 16-bit, the total of

XCOUNT * YCOUNT should be 1/2 (8-bit mode) or equal to (16-bit mode) the number of data reads or writes the host processor will perform.

Different interrupts are triggered upon completion of host data read and host data write DMA work units.

In order to synchronize work unit transitions, a separate interrupt is provided for host read operations. This interrupt will not be triggered until the host has read the final data in the work unit from the HOSTDP's FIFO. It is cleared by writing 1 to the HOSTRD_DONE bit in HOST_STATUS. A host data write operation will not trigger an interrupt until all data from the work unit has been written from the HOSTDP's FIFO to the specified location in memory. The interrupt is triggered on the selected DMA channel's assigned IRQ channel. It is cleared by writing 1 to the DMA_DONE bit in the appropriate DMAx_IRQ_STATUS register.

HOSTDP Modes of Operation

There are two modes of flow control in the HOSTDP—Acknowledge mode and Interrupt mode. These two modes provide flow control between the host and the slave processor by way of a single hardware signal. This signal has different names depending upon the mode of operation. The flow control mode is configured by the slave processor when enabling the HOSTDP (see HOST_CONTROL register).

In Acknowledge Mode, the signal is called HRDY and is used to add wait states to a host transaction when the HOSTDP isn't ready to transfer data. The HRDY signal is level-sensitive.

In Interrupt Mode, the signal is called FRDY and is used as an edge-triggered signal. This signal is connected to the host as an interrupt input. A falling edge on it signals to the host that the HOSTDP is ready for a guaranteed FIFO depth number of back to back transactions. For Host Write operations, this occurs when the FIFO is empty. For Host Read operations, this occurs when the FIFO is full.

Acknowledge Mode

For Host Data Write operations, HRDY will negate when the FIFO is full, thereby inserting wait states. As soon as the DMA engine moves data out of the FIFO, HRDY will assert, indicating to the host that the Host Data Write operation has completed.

For Host Data Read operations, HRDY will negate when the FIFO is empty, thereby inserting wait states. As soon as the DMA engine moves data into the FIFO, HRDY will assert, indicating to the host that the Host Data Read operation has completed.

The HRDY signal must be pulled high through an external pull-up resistor by default at power up/reset and when the HOSTDP is not enabled. HRDY is only driven when $\overline{\text{HOST\_CE}}$ is asserted low.

When the host is performing a host write configuration or HOST_STATUS reads, HRDY will always remain asserted and no wait states will be added.

Acknowledge Mode Timing Diagrams

This section gives further details on the HOSTDP timings for acknowledge mode. The host processor must follow these rules on every bus cycle, independent of the nature of the access and the status of slave processor.

It is assumed that the Blackfin slave processor has booted and the HOSTDP is functional.

As discussed in the following, HRDY has an external pull-up resistor:

1. If $\overline{\text{HOST\_CE}}$ is high, HRDY is three-state (not driven).
2. HRDY is driven by the slave processor (according to the rules shown below) only when $\overline{\text{HOST\_CE}}$ is asserted low by the external host device.

3. If HOST_CE and either HOST_RD or HOST_WR are asserted, and HOST_ADDR is high (configuration port access), HRDY will remain driven high.

4. If HOST_CE and either HOST_RD or HOST_WR are asserted, and HOST_ADDR is low (data port access), one of two things will happen:

    a. If HOST_RD is asserted and the desired FIFO data can be transferred on the data bus pins within time T, HRDY will remain driven high. If HOST_WR is asserted and the data can be stored in the FIFO within time T, HRDY will remain high.

    b. If the desired FIFO data cannot be transferred on the data bus pins or stored in the FIFO within time T, HRDY will be driven low quickly. Some time after the desired data operation is complete, HRDY is driven high.

The two timing diagrams, shown in Figure 8-3 and Figure 8-4, are necessary to understand the function of HRDY.

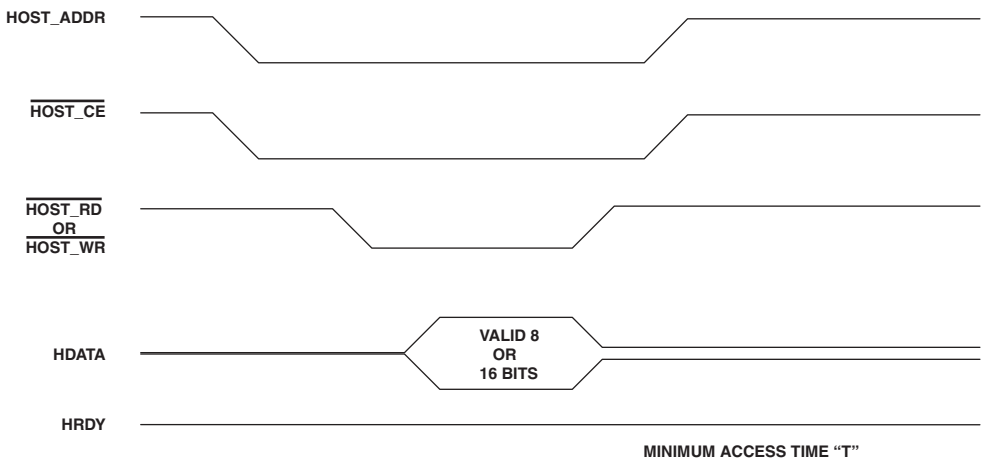


Figure 8-3. No delay in Host Bus Cycle

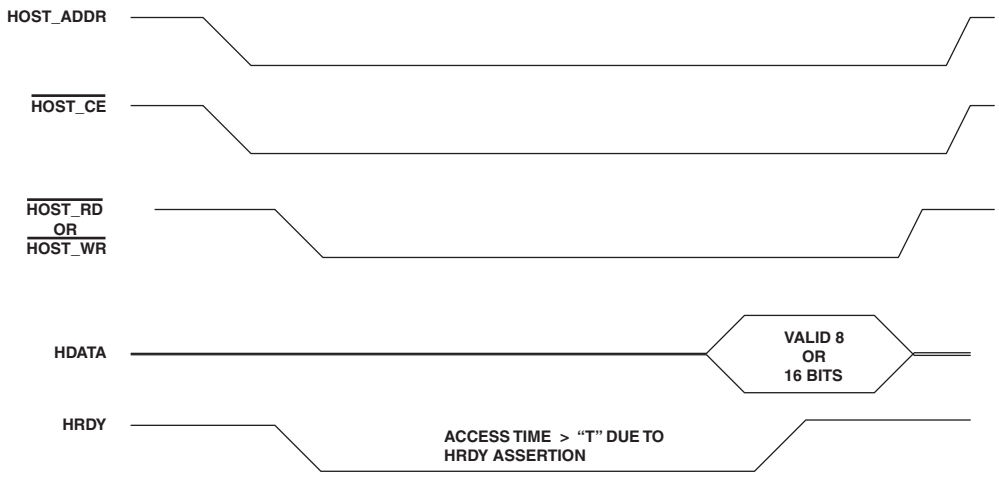


Figure 8-4. Delay in Host Bus Cycle caused by HRDY

Host Bus Timeout

In acknowledge mode, an optional Host Bus Timeout feature is implemented as a mechanism to alert the host when a programmed period of time has expired during a Host Read/Write Data transaction and the HOSTDP is still unable to complete the transaction with HRDY assertion. This condition can occur when the internal shared DMA bus has a lot of traffic from other peripherals on it. An internal timer is started when HOST_CE and either HOST_RD or HOST_WR are asserted. The timer is reset whenever HRDY is asserted.

This feature can be enabled by setting the BT_EN bit in the HOST_CONTROL register. When set, the HOSTDP will generate an interrupt when a pre-programmed time-out value in the HOST_TIMEOUT register expires. In a typical application the interrupt service routine toggles a GPIO pin, which is connected to the host processor to alert it of this condition. Addition-

ally, the interrupt service routine can perform writes to the HOST_CONTROL register to perform the following:

- Stop the DMA channel
- Assert the HRDY pin to allow the Host Bus cycles to continue while the host is being signaled of this condition by way of a GPIO pin
- Disable the HOSTDP

Because it's important for the host to be aware that a timeout condition occurred, it is required that the host processor read the HOST_STATUS register and check the HOSTDP_TOUT bit. The ADSP-BF52x slave processor reads the actual bit, allowing it to take the timeout interrupt, and write-one-to-clear the HOSTDP_TOUT bit. The host processor reads a special shadow version of this bit which will remain set until the host has read it or a hard reset occurs.

Interrupt Mode

The FRDY signal will act as an edge-sensitive (high-to-low transition) signal to provide an interrupt to the external host to indicate when data transfers can proceed. The interrupt provided by the slave processor to the external host device by way of the FRDY signal is used to indicate the status of the Host DMA Port's FIFO. Host Data Read and Host Data Write accesses are described next. The host device always masters the transactions and the Blackfin processor is always a DMA slave device.

In Interrupt Mode, the FRDY signal will always be driven by the slave processor and does not require an external pull-up resistor.

For Host Write operations, FRDY will transition from high to low whenever the FIFO is empty, causing an interrupt to the host to tell it to write to HOSTDP. The host can then perform a buffer depth number of write cycles to fill the FIFO. During these writes, FRDY will transition high again, but this is ignored by the host. After the FIFO's contents have been moved to memory by the DMA engine, the FIFO will become empty. At

this time, FRDY will once again transition from high to low to interrupt the host to do another buffer depth number of write cycles to fill the FIFO. This process continues until the configured number of words have been transferred.

For Host Read operations, FRDY will transition from high to low whenever the FIFO is full, causing an interrupt to the host to tell it to read from the HOSTDP. The host can then perform a buffer depth number of read cycles to empty the FIFO. During these reads, FRDY will transition high again, but this is ignored by the host. The DMA engine will fill the FIFO from data in memory. Once the FIFO becomes full again, FRDY will once again transition from high to low to interrupt the host to do another buffer depth number of write cycles to fill the FIFO. This process continues until the configured number of words have been transferred.

In interrupt mode, the FRDY signal always reflects the status of the FIFO. For Host Configuration writes or Host reads of HOST_STATUS, accesses will always meet the minimum cycle time T and FRDY will not be used for flow control of these accesses.

Figure 8-5 shows the timing of the interrupt mode transactions. The total number of words in the transfer are divided into blocks that contain a FIFO depth's number of words. These blocks are transferred whenever a high-to-low transition occurs on FRDY.

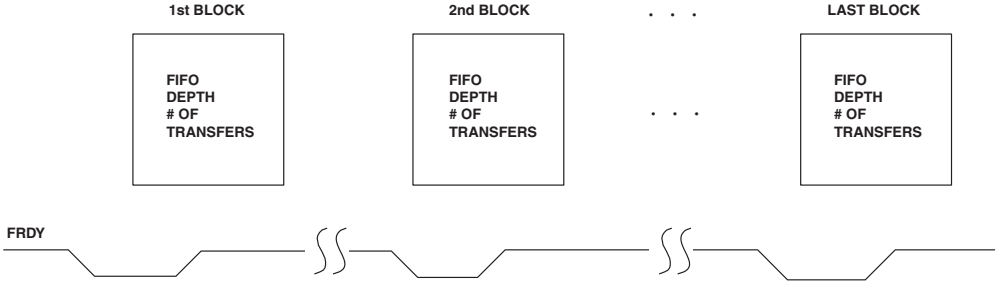


Figure 8-5. Interrupt Mode Bus Cycles

DMA STOP Mode and AUTOBUFFER Mode

The FLOW bit in HOST_CONFIG controls whether the DMA channel runs in stop mode or autobuffer mode.

In stop mode the DMA performs a block transfer once, as programmed by the HOST_CONFIG, XCOUNT/YCOUNT, XMODIFY/YMODIFY, and START_ADDR.L/H registers. Performing another block transfer requires the host to reconfigure these parameters. For stop mode, the interrupt service routine is required to set the DMA_CMPLT bit in the HOST_STATUS register. This prepares the HOSTDP for the next transfer. The host is not required to poll the DMA_CMPLT bit before starting a new work unit.

In autobuffer mode, the DMA performs continuous block transfers based on the parameters programmed by the HOST_CONFIG, XCOUNT/YCOUNT, XMODIFY/YMODIFY, and START_ADDR.L/H registers. Once the number of words specified by XCOUNT/YCOUNT are transferred, the DMA engine sets its address pointer back to START_ADDR.L/H and performs another block transfer. For autobuffer mode, the interrupt service routine should only set the DMA_CMPLT bit in the HOST_STATUS when it wishes to complete the transfers. After this bit is set, the HOSTDP block expects to be reprogrammed with a new set of DMA register values.

Bus Widths and Endian Order

The HOSTDP can be programmed to be either 16 bits wide or 8 bits wide. Additionally the byte order can be programmed as little endian or big endian. All ensuing data and configuration transactions with the host will occur in the programmed endian setting.

For 16-bit transfers, shown in Figure 8-6, the upper and lower bytes will be based on the big/little endian setting. When set to little endian, the order of the bytes on the HOST_DATA[15:0] bus is unchanged. For big

endian, the upper and lower bytes of HOST_DATA[15:0] are swapped before being stored internally.

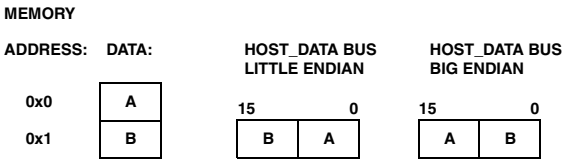


Figure 8-6. 16-bit Transfer Byte Order

For 8-bit transfers the order in which the bytes are sent will be based on the big/little endian setting as shown in Figure 8-7. Consider a 16-bit word stored in internal memory:

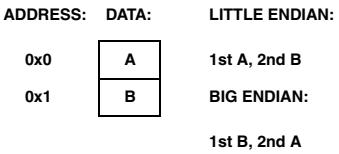


Figure 8-7. 8-bit Transfer Byte Order

Access Control

Configurations only occur when they are allowed. If the access type is disallowed, the ALLOW_CNFG bit will not go low after configuration words are written. In the case of a disallowed configuration, the configuration words do not show up on the DAB bus, the DMA controller does not get programmed, and no NACK is provided to the host.

By default, the HOSTDP module prohibits the external host from performing host data reads and writes. Blackfin software is required to enable host reads or writes. Host data reads and writes are enabled or disabled separately by the EHR and EHW bits in the HOST_CONTROL register. Once enabled, the host can perform read or write transactions. Writes to the

configuration port, control commands, and status reads are permitted regardless of the EHR and EHW settings.

For more information, see the memory configuration discussion in "Security" on page 16-1.

In acknowledge mode, if the transactions are disabled, host writes will still be allowed on the bus, but the actual write data is ignored. Similarly, host reads will still occur on the bus, but the data returned is indeterminate.

In interrupt mode, transitions on FRDY will never occur.

Note that the host cannot interrogate the HOSTDP to see whether only read or write access is granted. Therefore, it is recommended to keep EHR and EHW settings global without altering them.

Improving HOSTDP DMA Bus Bandwidth

Since the HOSTDP can be configured as a 16-bit wide parallel interface, data can move into and out of the peripheral quickly as compared to other serial peripherals on the chip. A FIFO is used to buffer this data and internal DMA bus requests are made judiciously to minimize the amount of DMA bandwidth that is used on the DMA bus. DAB bus arbitration overhead and direction change penalties are minimized. This is the default behavior (BDR=1). The HOSTDP follows the algorithm shown in Table 8-3, for receive (host write) operations.

Table 8-3. Host Write Operation

16-bit Words in FIFO	DMA Request Frequency (SCLK cycles)	Bursts per DMA Request
1 – 4	24	Up to 4
5 – 8	16	4
9 – 12	8	4
>12	2	0

For example if there are ten words written into the FIFO by the host processor DMA will be requested on the eighth SCLK cycle. Once the DAB approves the request, it will transfer four words. Assuming the host processor does not write any new words to the FIFO, the HOSTDP will request DMA 16 cycles again and another four words will be transferred. Twenty-four SCLK cycles later, the remaining two words are transferred. Note that words stored in the FIFO are 16-bit.

For transmit (host read) operation, the algorithm looks similar. Refer to Table 8-4.

Table 8-4. Host Read Operation

16-bit Words in FIFO	DMA Request Freq (SCLK cycles)	Bursts per DMA Request
0 – 4	2	0
5 – 8	8	4
9 – 12	16	4
>12	24	Up to 4

This default behavior can be overridden by clearing the Burst DMA Requests (BDR) bit in the HOST_CONTROL register. This is to allow the HOSTDP to perform internal DMA bus requests whenever there is a single word of data in the FIFO for host writes, and at least one empty slot for host reads. In this case DMA bus requests are made more often. This allows higher throughput through the HOSTDP at the expense of the other peripherals on the chip.

Control Commands Between the External Host and HOSTDP

Control commands can be sent from the host to the HOSTDP by writing to the configuration port with bits 3 and 2 of the data high. When the Host DMA port is waiting for configuration, a control command cannot be sent because it will be misinterpreted as a configuration write. After

configuration is finished, control commands can then be issued at any time. If the host is unsure of whether configuration is pending, it will need to read the HOST_STATUS register to check.

The commands that are supported are shown in Table 8-5.

Table 8-5. Control Commands

HOST_DATA[7:0]	Command
b#000111xx	HOST IRQ
b#001011xx	DMA FINISH
b#001111xx to b#111111xx	ignored

The HOST IRQ command provides a mechanism for the host to interrupt the HOSTDP. When the host writes a HOST IRQ command to the configuration port, the HIRQ bit in the HOST_STATUS register is set and a HOSTDP status interrupt is signaled.

The handshake bit (HSHK) in HOST_STATUS can be set or cleared anytime by the slave processor. This bit can be used as a flag that the host can read. In an application, the host might interrupt with the HOST IRQ command requesting information. The interrupt service routine could then set or clear the HSHK bit. The host could then read the status register and test for the value of the HSHK bit.

The DMA FINISH command performs the same functions as the HOSTDP reset (HOSTDP_RST) bit in the HOST_CONTROL register, except that it modifies the HOST_STATUS register contents and stops any DMA activity. The DMA FINISH command may not complete immediately but completes only after the DAB state machine has moved to a particular idle state.

There are additional restrictions on when a DMA FINISH command may be sent by the host processor, refer to "DMA Control Commands" on page 6-107.

When the HOSTDP module receives a DMA FINISH command from the host during a write operation, the DMA channel's FIFO is still drained gracefully and requests a DMA completion interrupt. However, the HOSTDP's FIFO is flushed immediately. To avoid loss of data, the host may want to wait until the FIFOEMPTY bit in HOST_STATUS is asserted before issuing the DMA FINISH command.

Programming Model

Figure 8-8 on page 8-22 and Figure 8-9 on page 8-23 show how to enable the Host DMA Port. They also show how to properly set up interrupt ser-

vice routines for both host read and write which clear the interrupts and prepare the HOSTDP to be configured by the host again.

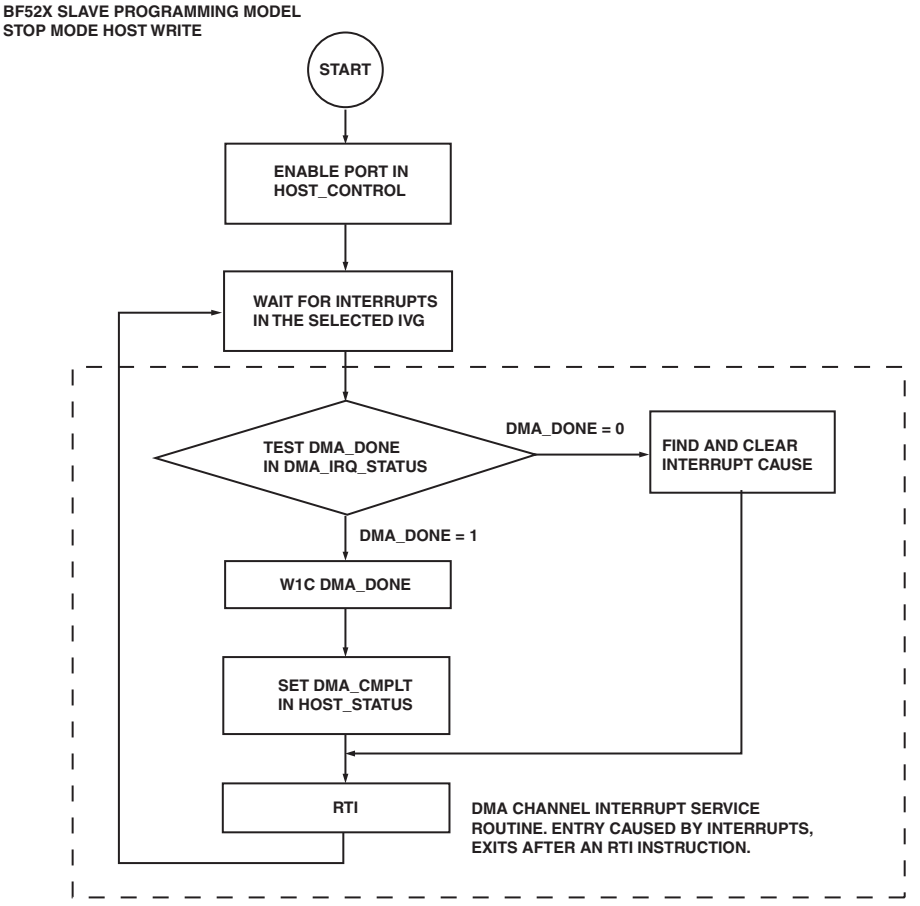


Figure 8-8. Stop Mode Host Write

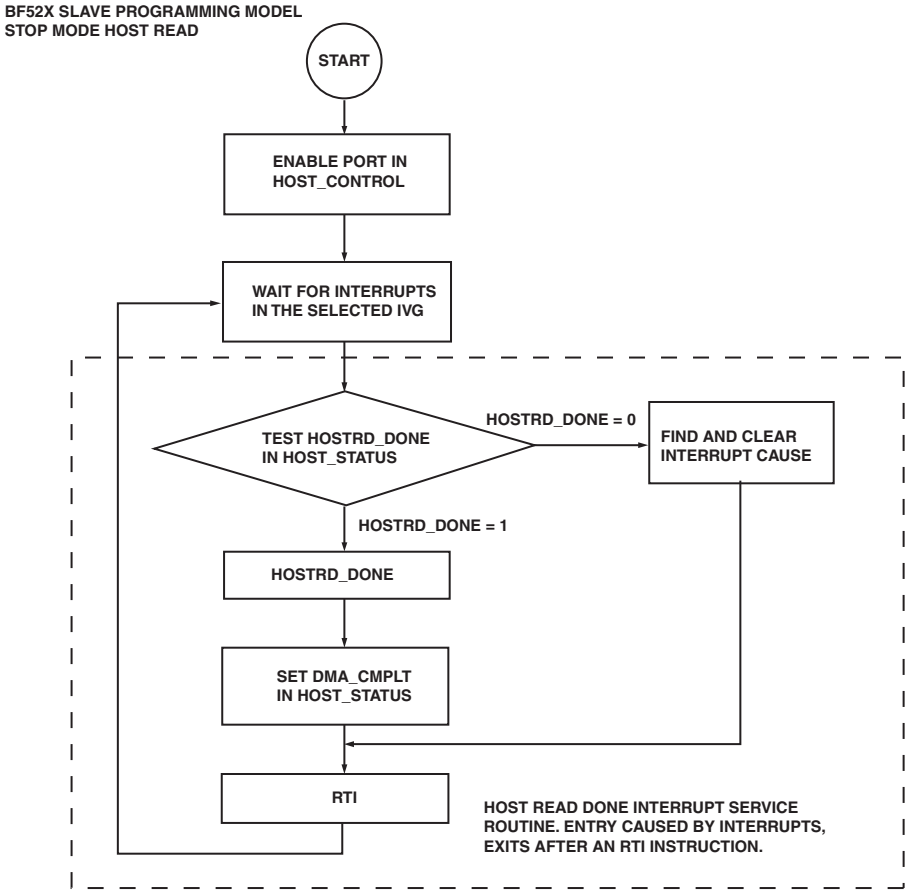


Figure 8-9. Stop Mode Host Read

Figure 8-10 on page 8-24 and Figure 8-11 on page 8-25 show how to program a host processor to send a configuration to the ADSP-BF52x slave.

They also show when to send data in both acknowledge and interrupt modes.

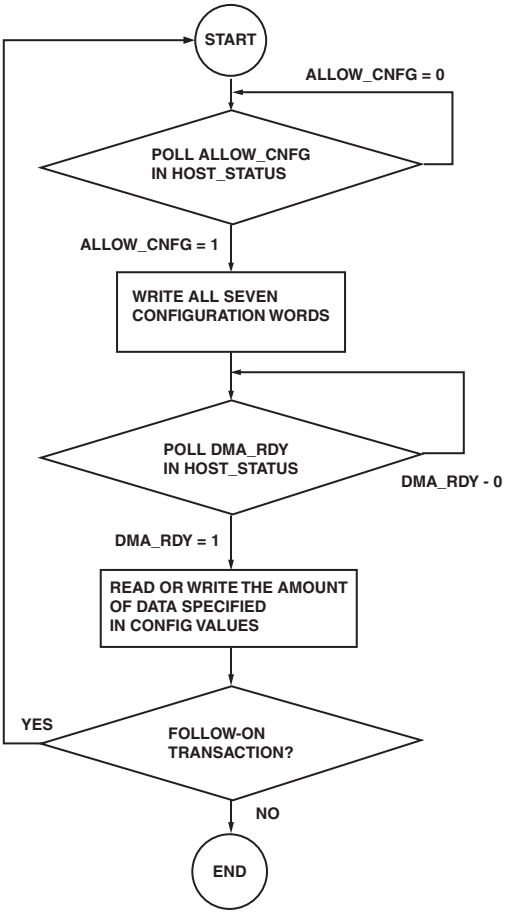


Figure 8-10. Acknowledge Mode with DMA Set to Stop Mode

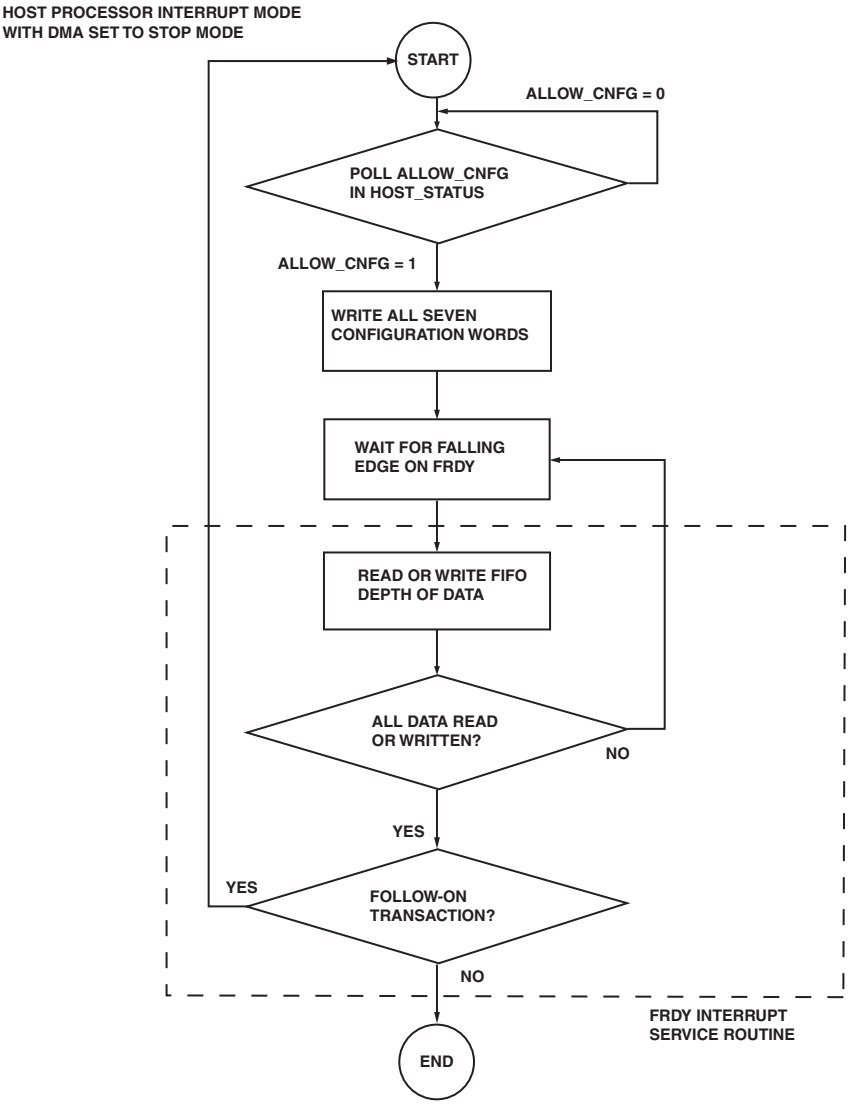


Figure 8-11. Interrupt Mode with DMA Set to Stop Mode

Host DMA Port Registers

Descriptions and bit diagrams for each of the MMRs discussed in this chapter are provided in Figure 8-12 through Figure 8-14.

HOSTDP Control (HOST_CONTROL) Register

The Host Control register (HOST_CONTROL), shown in Figure 8-12, is used to enable the HOSTDP module and to establish transfer modes of operation.

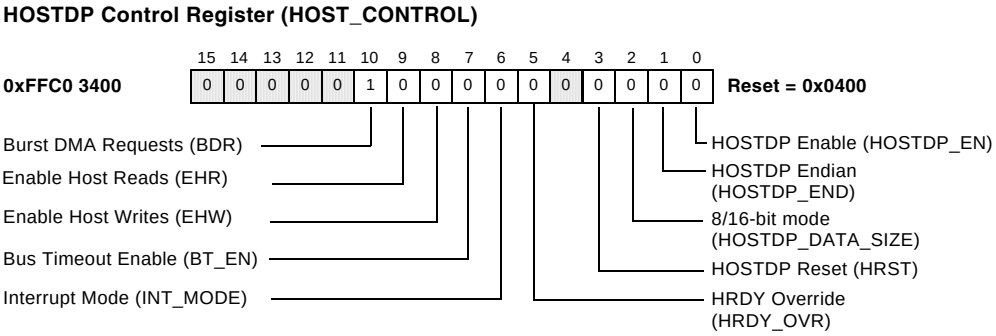

HOSTDP Control Register (HOST_CONTROL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 3400
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
Reset = 0x0400
Burst DMA Requests (BDR)
Enable Host Reads (EHR)
Enable Host Writes (EHW)
Bus Timeout Enable (BT_EN)
Interrupt Mode (INT_MODE)
HOSTDP Enable (HOSTDP_EN)
HOSTDP Endian (HOSTDP_END)
8/16-bit mode (HOSTDP_DATA_SIZE)
HOSTDP Reset (HRST)
HRDY Override (HRDY_OVR)

Figure 8-12. HOSTDP Control Register

Additional information for HOST_CONTROL register bits include:

- **HOSTDP Enable** (HOSTDP_EN)

  This bit enables the HOSTDP interface. It is used as a control signal to steer the PPI's resources to the HOSTDP. Always reset HOSTDP before disabling it.

- **Little/Big Endian** (HOSTDP_END)

When set, this bit swaps the lower and upper byte of data when reading or writing the HOSTDP FIFO. A value of 0 represents little endian and a value of 1 represents big endian.

- **8/16-bit Host Data Transfer** (HOSTDP_DATA_SIZE)

  This bit sets the HOSTDP external data transfer width. This bit, along with HOSTDP_EN, is used as a control signal to steer the PPI resources to the HOSTDP. A value of 0 is 8-bit data and a value of 1 is 16-bit

- **HOSTDP Reset** (HOSTDP_RST)

  This is a soft reset which does not affect the contents of HOST_CONTROL. Programming this bit causes the FIFO to flush, turns off the DMA channel, and returns the HOSTDP to a state where it waits for configuration. It also causes HOST_STATUS to clear to the same value as a hard reset with the exception of the BTE bit, which is always the same as BT_EN in HOST_CTL. A host DMA port reset does complete immediately, but completes only after the DAB state machine has moved to a particular idle state. This bit always reads as b#0.

- **HRDY Override** (HRDY_OVR)

  Setting this bit high will force HRDY high. When set, HRDY will be driven high for all remaining FIFO transfers and the ALLOW_CNFG bit will be driven low to prevent accidental configurations.

- **Interrupt Mode** (INT_MODE)

  When set this bit is used to select Interrupt mode. When cleared it selects Acknowledge Mode.

- **Bus Timeout Enable** (BT_EN)

When set this bit enables the HOSTDP interrupt to occur when a current host transaction has not finished before a programmed timeout value occurs.

- **Enable HOSTDP Write** (EHW)

  When set this bit enables HOSTDP writes to occur. When cleared, host writes appear to occur on the pins, but the actual write data is ignored.

- **Enable HOSTDP Read** (EHR)

  When set this bit enables HOSTDP reads to occur. When cleared, host reads return zero data.

- **Burst DMA Requests** (BDR)

  When set, as by default, the HOSTDP module groups multiple data words and requests DMA bursts to the DAB bus. When cleared, every individual data word requests a separate DMA transfer.

HOSTDP Status (HOST_STATUS) Register

The HOSTDP status register (HOST_STATUS), shown in Figure 8-13, holds the key status information of the HOSTDP. Bits in this register are read by the external host to query the status of the transaction. This register

can also be read and written through the PAB. Note the differences in how to write and clear bits as well as the many bits which are read-only.

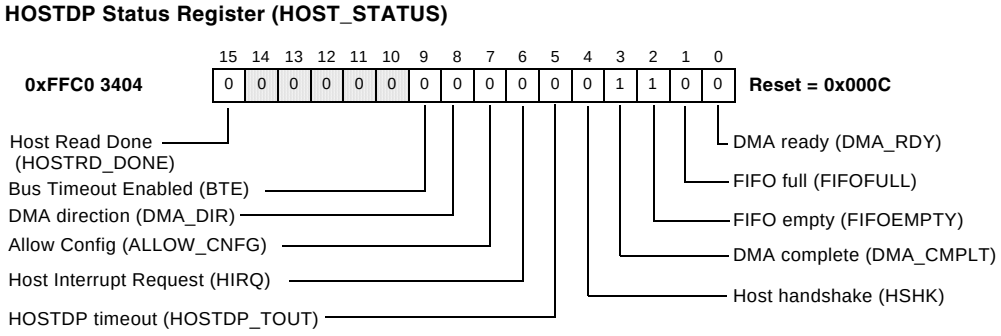

HOSTDP Status Register (HOST_STATUS)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 3404
0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
Reset = 0x000C
Host Read Done (HOSTRD_DONE)
Bus Timeout Enabled (BTE)
DMA direction (DMA_DIR)
Allow Config (ALLOW_CNFG)
Host Interrupt Request (HIRQ)
HOSTDP timeout (HOSTDP_TOUT)
DMA ready (DMA_RDY)
FIFO full (FIFOFULL)
FIFO empty (FIFOEMPTY)
DMA complete (DMA_CMPLT)
Host handshake (HSHK)

Figure 8-13. HOSTDP Status Register

Additional information for the HOST_STATUS register bits include:

- **DMA ready** (DMA_RDY) - Read-Only

  This bit is set one cycle after the last control word (YMODIFY) is written to the DMA. The bit is cleared when the DMA_CMPLT bit is set by software.

- **FIFO full** (FIFOFULL) - Read-Only

  This bit is set when the HOSTDP FIFO is full.

- **FIFO empty** (FIFOEMPTY) - Read-Only

  This bit is set when the HOSTDP FIFO is empty.

- **DMA complete** (DMA_CMPLT) - Write-1-to-set

  This bit must be set by software in the interrupt service routine called when the DMA operation is completed. This bit is cleared after the last control word (YMODIFY) is written to the DMA controller.

- **HOSTDP handshake** (HSHK) - Read/Write

  This bit is set and cleared by software and functions as a general-purpose handshake bit. It is often used to indicate an error to the host device. This bit does not control HOSTDP hardware and is cleared by the HOSTDP_RST bit.

- **HOSTDP timeout** (HOSTDP_TOUT) - Write-one-to-clear

  This bit is set when the HOSTDP time-out occurs. When set, it requests a HOSTDP status interrupt. The interrupt service routine (ISR) must clear this bit.

- **Allow Configurations** (ALLOW_CNFG) - Read-Only

  The host processor is required to poll this bit to see when the Host DMA port has been enabled and configuration writes are allowed. This bit is cleared when the last configuration word (YMODIFY) is written by the host. The bit is set again when the descriptor has been completely passed to the DMA channel.

- **HOSTDP Interrupt Request** (HIRQ) - write-1-to-clear

  This bit is set when the host writes a HOSTDP IRQ control command to the configuration port. When set, this bit requests a HOSTDP status interrupt. The interrupt service routine (ISR) must write this bit to one to clear it.

- **DMA direction** (DMA_DIR) - Read-Only

  This bit is cleared for read DMA and set for write DMA. It reflects the WNR bit in the DMA_CONFIG word. If a former work unit was active, the bit does not update until the DMA_CMPLT bit is set by software.

- **Bus Timeout Enabled** (BTE) - Read-Only

This bit is just a copy of the BT_EN bit in the HOST_CONTROL register. The host can read this bit to determine if software has enabled the Bus Timeout feature.

This bit must be set by the interrupt service routine software which is called when the DMA is finished.

- **Host Read Done** (HOSTRD_DONE) - Write-one-to-clear

  This bit is set when a Host Data read DMA work unit has completed. It must be cleared by software in the associated interrupt service routine.

HOSTDP Timeout (HOST_TIMEOUT) Register

The Host Bus Timeout feature is previously described in "Acknowledge Mode" on page 8-11.

HOSTDP Timeout Register (HOST_TIMEOUT)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0xFFC0 3408	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**Reset = 0x0000**

Figure 8-14. HOSTDP Timeout Register

The HOSTDP Time-out register (HOST_TIMEOUT), shown in Figure 8-14, holds the time-out value. A timer is loaded with this value when a host transaction is started. If HOSTDP doesn't respond with HRDY within the programmed amount of time, the HOSTDP_TOUT bit in the HOST_STATUS register is set and an interrupt is generated. This feature takes effect only when the BT_EN bit in the HOST_CONTROL register is set to 1.

The length of the timeout generated by this register is governed by the following equation:

$$timeout = (2\text{^}16 * HOST\_TIMEOUT)/(sclk)$$

For example, using an SCLK frequency of 133 MHz and HOST_TIMEOUT = 0x7ED, the timeout period is approximately one second.

Programming Examples

Information for this section will be added when it becomes available.

9 GENERAL-PURPOSE PORTS

This chapter describes the general-purpose ports. Following an overview and a list of key features is a block diagram of the interface and a description of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Overview

The ADSP-BF52x Blackfin processors feature a rich set of peripherals, which through a powerful pin multiplexing scheme, provides great flexibility to the external application space.

Table 9-1shows all the peripheral signals that can be accessed off chip.

Table 9-1. General-Purpose and Special Function Signals

Peripheral	Signals
10/100 Ethernet MAC[1]	MII interface (18) or RMII (11)
Host DMA	Data (16), control (5))
TWI Controller	Data (1), clock (1)
PPI Interface	Data (16), frame sync (3), clock (1)
SPI Interface	Data (2), clock (1), slave select (1), slave enable (7)
SPORTs	Data (8), clock (4), frame sync (4)
UARTs	Data (4)
Timers	PWM/capture/clock (8), alternate clock input (4), alternate capture input (7)

Table 9-1. General-Purpose and Special Function Signals (Continued)

Peripheral	Signals
General-Purpose I/O	GPIO (48)
Handshake MemDMA	MemDMA request (2)

1 ADSP-BF526 and ADSP-BF527 only.

Features

The peripheral pins are functionally organized into general-purpose ports designated port F, port G, port H, and port J.

Port F provides 16 pins:

- UART1 signals
- PPI data signals
- Up/Down Counter
- SPORT0 and SPORT1 signals
- NFC data signals
- Alternate timer inputs
- Additional SPI slave selects
- GPIOs

Port G provides 16 pins:

- SPORT0 signals
- Primary SPI signals
- UART0 and UART1 signals

- Host DMA control signals
- Handshake memDMA request signals
- Primary timer signals
- MII/RMII pins
- GPIOs

Port H provides 16 pins:

- MII/RMII signals (ADSP-BF526 and ADSP-BF527 processors only)
- Alternate timer inputs
- Additional SPI slave selects
- Host DMA
- NFC control and data signals
- GPIOs

Port J provides 4 pins:

- TWI signals
- PPI clock and frame sync signals
- Timer clock and primary timer signals

Interface Overview

By default, all pins of port F, port G, and port H are in general-purpose I/O (GPIO) mode. Port J does not provide GPIO functionality. In this mode, a pin can function as either digital input, digital output, or interrupt input. See “General-Purpose I/O Modules” on page 9-13 for details.

Peripheral functionality must be explicitly enabled by the function enable registers (PORTF_FER, PORTG_FER, and PORTH_FER). The competing peripherals on port F, port G, and port H are controlled by the respective multiplexer control register (PORTF_MUX, PORTG_MUX, PORTH_MUX).

In this chapter, the naming convention for registers and bits uses a lower case x to represent F, G, or H. For example, the name PORTx_FER represents PORTF_FER, PORTG_FER, and PORTH_FER. The bit name PxO represents PF0, PG0, and PH0. This convention is used to discuss registers common to these three ports.

External Interface

The external interface of the general-purpose ports are described in the following sections.

Port F Structure

Table 9-2 on page 9-5 shows the multiplexer scheme for port F. Port F is controlled by the PORTF_MUX and the PORTF_FER registers.

Port F consists of 16 pins, referred to as PF0 to PF15, as shown in Table 9-2 on page 9-5. Besides the 16 GPIOs, this port supports all SPORT0 and SPORT1 signals. If the secondary data pins are not needed, the corresponding pins can be used for GP Timer purposes. SPORT1 pins are multiplexed with PPI data signals PPID15-8. Thus, with an 8-bit PPI configuration, no restrictions apply to SPORT1. All the input signals in the Additional Use column are enabled by their module only, regardless of the state of PORTx_MUX and PORTx_FER registers.

Any GPIO can be enabled individually and overrides the peripheral function if the respective bit in the PORTF_FER is cleared.

Table 9-2. Port F Multiplexing Scheme

PORTF_MUX	00	01	10	11		
	1st Function	2nd Function	3rd Function	4th Function	Additional Use	GPIO
Bit[1:0]	PPI D0 PPI D1 PPI D2 PPI D3 PPI D4 PPI D5 PPI D6 PPI D7	DR0PRI RFS0 RSCLK0 DT0PRI TFS0 TSCLK0 DT0SEC DR0SEC	NFC D0 NFC D1 NFC D2 NFC D3 NFC D4 NFC D5 NFC D6 NFC D7	- - - - - - - -	 TACLK0 TACLK1 TACI0 TACI1	PF0 PF1 PF2 PF3 PF4 PF5 PF6 PF7
Bit[3:2]	PPI D8 PPI D9	DR1PRI RSCLK1	 SPI_SSEL6	- -		PF8 PF9
Bit[5:4]	PPI D10	RFS1	SPI_SSEL7	-		PF10
Bit[7:6]	PPI D11	TFS1		-	CZM	PF11
Bit[9:8]	PPI D12 PPI D13	DT1PRI TSCLK1	SPI SSEL2 SPI SSEL3	- -	CDG CUD	PF12 PF13
Bit[11:10]	PPI D14 PPI D15	DT1SEC DR1SEC	UART1TX UART1 RX	- -	 TACI3	PF14 PF15

Bit 12 of the PORTF_MUX register controls the input enable for the PPICLK/TMRCLK pin available in Port J. If bit 12 is set to 1, the PPICLK/TMRCLK is enabled. If bit 12 is set to 0, the PPICLK/TMRCLK is disabled. Bits 13-15 in the PORTF_MUX register are reserved.

Port G Structure

Table 9-3 on page 9-6 shows the multiplexer scheme for port G. It is controlled by the PORTG_MUX and the PORTG_FER registers.

Port G consists of 16 pins, referred to as PG0 to PG15, as shown in Table 9-3. Besides the 16 GPIOs, this port supports both UART0 and

UART1 signals along with Host DMA control signals. If only one UART is required in the target application, the user has the option to enable either two additional timers or the handshake memDMA request pins. For more information, see “Handshaked Memory DMA Operation” in Chapter 6, Direct Memory Access.

Table 9-3. Port G Multiplexing Scheme

PORTG_MUX	00	01	10	11		
	1st Function	2nd Function	3rd Function	4th Function	Additional Use	GPIO
				- -		PG0 (HWAIT)
Bit[1:0]	SPI SS SPI SCK SPI MISO SPI MOSI	 DR0SEC DT0SEC	SPI SSEL1 SPI SCK SPI MISO SPI MOSI	- - - -		PG1 PG2 PG3 PG4
Bit[3:2]	TMR1/ PPI FS2 DT0PRI	 TMR2	TMR1/ PPI FS2 PPI FS3	- -		PG5 PG6
Bit[5:4]	TMR3 TMR4	DR0PRI RFS0	UART0 TX UART0 RX	- -	 TACI4	PG7 PG8
Bit[7:6]	TMR5	RSCLK0		-	TACI5	PG9
Bit[9:8]	TMR6	TSCLK0		-	TACI6	PG10
Bit[11:10]	TMR7 DMAR1 DMAR0	TMR7 UART1TX UART1RX	HOST WR HOST ACK HOST ADDR	- -	 TACI2	PG11 PG12 PG13
Bit[13:12]	TSCLK0 TFS0	RMII MDC RMII PHYINT	HOST RD HOST_CE	- -		PG14 PG15

When TMR6 is an output, SPORT0 ignores the external TSCLK0 signal on PG14.

Special attention is required for the use of the timers with PPI enabled. Timer 0 and Timer 1 are typically used for PPI frame sync generation.

Any GPIO can be enabled individually and overrides the peripheral function if the respective bit in the PORTG_FER is cleared.

Bits 14-15 in the PORTG_MUX register are reserved.

Port H Structure

Figure 9-4 shows the multiplexer scheme for port H. It is controlled by the PORTH_MUX and the PORTH_FER registers.

Port H consists of 16 pins, referred to as PH0 to PH15, as shown in Figure 9-4. Besides the 16 GPIOs, this port supports MII/RMII signals (ADSP-BF526 and ADSP-BF527 processors only) along with Host DMA signals. This port also contains the NFC control and data signals.

Any GPIO can be enabled individually and overrides the peripheral function if the respective bit in the PORTH_FER is cleared.

Table 9-4. Port H Multiplexing Scheme

PORTH_MUX	00	01	10	11		
	1st Function	2nd Function	3rd Function	4th Function	Additional Use	GPIO
Bit[1:0]	NFC D0	RMII CRS	HOST D0	-		PH0
	NFC D1	RMII ERxER	HOST D1	-		PH1
	NFC D2	RMII MDIO	HOST D2	-		PH2
	NFC D3	RMII	HOST D3	-		PH3
	NFC D4	ETxEN	HOST D4	-		PH4
	NFC D5	RMII	HOST D5	-		PH5
	NFC D6	TxCLK	HOST D6	-		PH6
	NFC D7	RMII ETxD0	HOST D7	-		PH7
		RMII ERxD0				
		RMII ETxD1				
Bit[3:2]	SPI SSEL4	RMII ERxD1	HOST D8	-	TACLK2	PH8
Bit[5:4]	SPI SSEL5	MII ETxD2	HOST D9	-	TACLK3	PH9
	ND_CE	MII ERxD2	HOST D10	-		PH10
	ND_WE_B	MII ETxD3	HOST D11	-		PH11
	ND_RE_B	MII ERxD3	HOST D12	-		PH12
	ND_BUSY	MII ERxCLK	HOST D13	-		PH13
	ND_CLE	MII ERxDV	HOST D14	-		PH14
	ND_ALE	MII COL	HOST D15	-		PH15

Bits 6-15 in the PORTH_MUX register are reserved.

Port J Structure

Figure 9-5 shows the multiplexer scheme for port J. Port J does not provide GPIO functionality. This port contains TWI signals.

Table 9-5. Port J Multiplexing Scheme

PJ0: TMR0/PPI FS1
PJ1: PPI CLK/TMRCLK
PJ2: SCL
PJ3: SDA

The PPICLK/TMRCLK pin is enabled by setting bit 12 in the PORTF_MUX register.

Input Tap Considerations

Input taps are shown in Table 9-2 on page 9-5, Table 9-3 on page 9-6, and Table 9-4 on page 9-8 under the "Additional Use" column. When input taps (as well as GPIO based taps) are used with other functionality enabled on the GPIO pins, the signals seen by the input tap modules might be different from what is seen on the pins. This is because different pin functions have different signal requirements with respect to when the signal is latched, if at all. Because of this, input taps multiplexed on certain pins may behave differently than those on other pins, depending on which pin function is selected. The input taps will see different signals than at the pins in the following cases:

- All GPIO inputs except PG0, PG1, PG2, PG9, PG10, PG11, PG12, PG13, PG14, PH3, PH4, PH5, PH7, PH9, PH11, PH13, PH15 when GPIO is tapped with PORTx_FER set to 1.
- TACLK0 if PORTF_FER[4] = 1 and PORTF_MUX[1:0] = b#00 or b#01
- TACLK1 if PORTF_FER[5] = 1 and PORTF_MUX[1:0] = b#00

- TACI0 if PORTF_FER[6] = 1 and PORTF_MUX[1:0] = b#00 or b#01
- TACI1 if PORTF_FER[7] = 1 and PORTF_MUX[1:0] = b#00 or b#01
- CZM if PORTF_FER[11] = 1 and PORTF_MUX[7:6] = b#00 or b#01
- CDG if PORTF_FER[12]= 1 and PORTF_MUX[9:8] = b#00 or b#01
- CUD if PORTF_FER[13] = 1 and PORTF_MUX[9:8] = b#00
- TACI3 if PORTF_FER[15] = 1 and PORTF_MUX[11:10] = b#00 or b#01
- TACI4 if PORTG_FER[8] = 1 and PORTG_MUX[5:4] = b#01
- TACLK2 if PORTH_FER[8] = 1 and PORTH_MUX[3:2] = b#01

Internal Interfaces

Port control and GPIO registers are part of the system memory-mapped registers (MMRs). The addresses of the GPIO module MMRs appear in Appendix B. Core access to the GPIO configuration registers is through the system bus.

The PORTx_MUX register controls the muxing schemes of port F, port G and port J.

The function enable register (PORTF_FER, PORTG_FER, PORTH_FER) enables the peripheral functionality for each individual pin of port x.

Internal Signals

- TACLK7 and TACLK6 connect to buffered USB clock internally
- TACLK5 and TACLK4 connect to CLKBUF signal internally
- TACI7 connects to the COUNTER0 TO output internally

- TMR0 is internally looped back to PPI_FS1 (to be used as internally generated frame sync). In this case, PPI_CLK is the clock input for the timer 0 module
- TMR1 is internally looped back to PPI_FS2 (to be used as internally generated frame sync) In this case, PPI_CLK is the clock input for the timer 1 module
- TACI3 (or TACI2) and TACI4 can be used for autobaud detection of UART
- PPI_CLK/TMRCLK can be used as a clock input for any of the timers
- If TMR5 is output and PORTG_MUX[7:6] == 00 and the RSCLK0 input enable is active, TMR5 is the clock input for RSCLK0
- If RSCLK0 is output and PORTG_MUX[7:6] == 01 and the TMR5 input enable is active, RSCLK0 is the clock input for TMR5
- If TACI5 is selected for the timer 5 module, then the signal from the PG9 pin is fed to both RSCLK0 and TACI5
- If TMR6 is output and PG_MUX[9:8] == 00, and the TSCLK0 input enable is active, then TMR6 is the clock input for TSCLK0.
- If TSCLK0 is output and PG_MUX[9:8] == 01 and the TMR6 input enable is active, then TSCLK0 is the clock input for TMR6.
- If TACI6 is selected for the timer 6 module, the signal from the PG10 pin is fed to both TACI6 and TSCLK0.

Performance/Throughput

The PFx, PGx, and PHx pins are synchronized to the system clock (SCLK). When configured as outputs, the GPIOs can transition at every other SCLK cycle.

When configured as inputs, the overall system design should take into account the potential latency between the core and system clocks. Changes in the state of port pins have a latency of 3 SCLK cycles before being detectable by the processor. When configured for level-sensitive interrupt generation, there is a minimum latency of 4 SCLK cycles between the time the signal is asserted on the pin and the time that program flow is interrupted. When configured for edge-sensitive interrupt generation, an additional SCLK cycle of latency is introduced, giving a total latency of 5 SCLK cycles between the time the edge is asserted and the time that the core program flow is interrupted.

Description of Operation

The operation of the general-purpose ports is described in the following sections.

Operation

The GPIO pins on port F, port G, and port H can be controlled individually by the function enable registers (PORTx_FER). With a control bit in these registers cleared, the peripheral function is fully decoupled from the pin. It functions as a GPIO pin only. To drive the pin in GPIO output mode, set the respective direction bit in the PORTxIO_DIR register. To make the pin a digital input or interrupt input, enable its input driver in the PORTxIO_INEN register.

By default all peripheral pins are configured as inputs after reset. port F, port G, and port H pins are in GPIO mode. However, GPIO input drivers are disabled to minimize power consumption and any need of external pulling resistors.

When the control bit in the function enable registers (PORTx_FER) is set, the pin is set to its peripheral functionality and is no longer controlled by the GPIO module. However, the GPIO module can still sense the state of

the pin. When using a particular peripheral interface, pins required for the peripheral must be individually enabled. Keep the related function enable bit cleared if a signal provided by the peripheral is not required by your application. This allows it to be used in GPIO mode.

General-Purpose I/O Modules

The processor supports 48 bidirectional or general-purpose I/O (GPIO) signals. These 48 GPIOs are managed by three different GPIO modules, which are functionally identical. One is associated with port F, one with port G, and one with port H. Every module controls 16 GPIOs available through the pins PF15-0, PG15-0, and PH15-0.

Each GPIO can be individually configured as either an input or an output by using the GPIO direction registers (PORTxIO_DIR).

When configured as output, the GPIO data registers (PORTFIO, PORTGIO, and PORTHIO) can be directly written to specify the state of the GPIOs.

The GPIO direction registers are read-write registers with each bit position corresponding to a particular GPIO. A logic 1 configures a GPIO as an output, driving the state contained in the GPIO data register if the peripheral function is not enabled by the function enable registers. A logic 0 configures a GPIO as an input.

Note when using the GPIO as an input, the corresponding bit should also be set in the GPIO input enable register. Otherwise, changes at the input pins will not be recognized by the processor.

The GPIO input enable registers (PORTFIO_INEN, PORTGIO_INEN, and PORTHIO_INEN) are used to enable the input buffers on any GPIO that is being used as an input. Leaving the input buffer disabled eliminates the

need for pull-ups and pull-downs when a particular PFx, PGx, or PHx pin is not used in the system. By default, the input buffers are disabled.

Once the input driver of a GPIO pin is enabled, the GPIO is not allowed to operate as an output anymore. Never enable the input driver (by setting PORTxIO_INEN bits) and the output driver (by setting PORTxIO_DIR bits) for the same GPIO.

A write operation to any of the GPIO data registers sets the value of all GPIOs in this port that are configured as outputs. GPIOs configured as inputs ignore the written value. A read operation returns the state of the GPIOs defined as outputs and the sense of the inputs, based on the polarity and sensitivity settings, if their input buffers are enabled. Table 9-6 helps to interpret read values in GPIO mode, based on the settings of the PORTxIO_POLAR, PORTxIO_EDGE, and PORTxIO_BOTH registers.

Table 9-6. GPIO Value Register Pin Interpretation

POLAR	EDGE	BOTH	Effect of MMR Settings
0	0	X	Pin that is high reads as 1; pin that is low reads as 0
0	1	0	If rising edge occurred, pin reads as 1; otherwise, pin reads as 0
1	0	X	Pin that is low reads as 1; pin that is high reads as 0
1	1	0	If falling edge occurred, pin reads as 1; otherwise, pin reads as 0
X	1	1	If any edge occurred, pin reads as 1; otherwise, pin reads as 0

For GPIOs configured as edge-sensitive, a readback of 1 from one of these registers is sticky. That is, once it is set it remains set until cleared by user code. For level-sensitive GPIOs, the pin state is checked every cycle, so the readback value will change when the original level on the pin changes.

The state of the output is reflected on the associated pin only if the function enable bit in the PORTx_FER register is cleared.

Write operations to the GPIO data registers modify the state of all GPIOs of a port. In cases where only one or a few GPIOs need to be changed, the user may write to the GPIO set registers, PORTxIO_SET, the GPIO clear registers, PORTxIO_CLEAR, or to the GPIO toggle registers, PORTxIO_TOGGLE instead.

While a direct write to a GPIO data register alters all bits in the register, writes to a GPIO set register can be used to set a single or a few bits only. No read-modify-write operations are required. The GPIO set registers are write-1-to-set registers. All 1s contained in the value written to a GPIO set register sets the respective bits in the GPIO data register. The 0s have no effect. For example, assume that PF0 is configured as an output. Writing 0x0001 to the GPIO set register drives a logic 1 on the PF0 pin without affecting the state of any other PFx pins. The GPIO set registers are typically also used to generate GPIO interrupts by software. Read operations from the GPIO set registers return the content of the GPIO data registers.

The GPIO clear registers provide an alternative port to manipulate the GPIO data registers. While a direct write to a GPIO data register alters all bits in the register, writes to a GPIO clear register can be used to clear individual bits only. No read-modify-write operations are required. The clear registers are write-1-to-clear registers. All 1s contained in the value written to the GPIO clear register clears the respective bits in the GPIO data register. The 0s have no effect. For example, assume that PF4 and PF5 are configured as outputs. Writing 0x0030 to the PORTFIO_CLEAR register drives a logic 0 on the PF4 and PF5 pins without affecting the state of any other PFx pins.

If an edge-sensitive pin generates an interrupt request, the service routine must acknowledge the request by clearing the respective GPIO latch. This is usually performed through the clear registers.

Read operations from the GPIO clear registers return the content of the GPIO data registers.

The GPIO toggle registers provide an alternative port to manipulate the GPIO data registers. While a direct write to a GPIO data register alters all bits in the register, writes to a toggle register can be used to toggle individual bits. No read-modify-write operations are required. The GPIO toggle registers are write-1-to-toggle registers. All 1s contained in the value written to a GPIO toggle register toggle the respective bits in the GPIO data register. The 0s have no effect. For example, assume that PG1 is configured as an output. Writing 0x0002 to the PORTGIO_TOGGLE register changes the pin state (from logic 0 to logic 1, or from logic 1 to logic 0) on the PG1 pin without affecting the state of any other PGx pins. Read operations from the GPIO toggle registers return the content of the GPIO data registers.

The state of the GPIOs can be read through any of these data, set, clear, or toggle registers. However, the returned value reflects the state of the input pin only if the proper input enable bit in the PORTxIO_INEN register is set. Note that GPIOs can still sense the state of the pin when the function enable bits in the PORTx_FER registers are set.

Since function enable registers and GPIO input enable registers reset to zero, no external pull-ups or pull-downs are required on the unused pins of port F, port G, and port H.

GPIO Interrupt Processing

Each GPIO can be configured to generate an interrupt. The processor can sense up to 48 asynchronous off-chip signals, requesting interrupts through five interrupt channels. To make a pin function as an interrupt pin, the associated input enable bit in the PORTxIO_INEN register must be set. The function enable bit in the PORTx_FER register is typically cleared. Then, an interrupt request can be generated according to the state of the pin (either high or low), an edge transition (low to high or high to low), or on both edge transitions (low to high and high to low). Input sensitivity is defined on a per-bit basis by the GPIO polarity registers (PORTFIO_POLAR, PORTGIO_POLAR, and PORTHIO_POLAR), and the GPIO interrupt sensitivity registers (PORTFIO_EDGE, PORTGIO_EDGE, and PORTHIO_EDGE). If configured

for edge sensitivity, the GPIO set on both edges registers (PORTFIO_BOTH, PORTGIO_BOTH, and PORTHIO_BOTH) let the interrupt request generate on both edges.

The GPIO polarity registers are used to configure the polarity of the GPIO input source. To select active high or rising edge, set the bits in the GPIO polarity register to 0. To select active low or falling edge, set the bits in the GPIO polarity register to 1. This register has no effect on GPIOs that are defined as outputs. The contents of the GPIO polarity registers are cleared at reset, defaulting to active high polarity.

The GPIO interrupt sensitivity registers are used to configure each of the inputs as either a level-sensitive or an edge-sensitive source. When using an edge-sensitive mode, an edge detection circuit is used to prevent a situation where a short event is missed because of the system clock rate. The GPIO interrupt sensitivity register has no effect on GPIOs that are defined as outputs. The contents of the GPIO interrupt sensitivity registers are cleared at reset, defaulting to level sensitivity.

The GPIO set on both edges registers are used to enable interrupt generation on both rising and falling edges. When a given GPIO has been set to edge-sensitive in the GPIO interrupt sensitivity register, setting the respective bit in the GPIO set on both edges register to both edges results in an interrupt being generated on both the rising and falling edges. This register has no effect on GPIOs that are defined as level-sensitive or as outputs. See Table 9-6 on page 9-14 for information on how the GPIO set on both edges register interacts with the GPIO polarity and GPIO interrupt sensitivity registers.

Each of the three GPIO modules provides two independent interrupt channels. Identical in functionality, these are called interrupt A and interrupt B. Both interrupt channels have their own mask register which lets you assign the individual GPIOs to none, either, or both interrupt channels.

Since all mask registers reset to zero, none of the GPIOs is assigned any interrupt by default. Each GPIO represents a bit in each of these registers. Setting a bit means enabling the interrupt on this channel.

Interrupt A and interrupt B operate independently. For example, writing 1 to a bit in the mask interrupt A register does not affect interrupt channel B. This facility allows GPIOs to generate GPIO interrupt A, GPIO interrupt B, both GPIO interrupts A and B, or neither.

A GPIO interrupt is generated by a logical OR of all unmasked GPIOs for that interrupt. For example, if PF0 and PF1 are both unmasked for GPIO interrupt channel A, GPIO interrupt A will be generated when triggered by PF0 or PF1. The interrupt service routine must evaluate the GPIO data register to determine the signaling interrupt source. Figure 9-1 illustrates the interrupt flow of any GPIO module's interrupt A channel.

When using either rising or falling edge-triggered interrupts, the interrupt condition must be cleared each time a corresponding interrupt is serviced by writing 1 to the appropriate bit in the GPIO clear register.

At reset, all interrupts are masked and disabled.

Similarly to the GPIOs themselves, the mask register can either be written through the GPIO mask data registers (PORTxIO_MASKA, PORTxIO_MASKB) or be controlled by the mask A/mask B set, clear and toggle registers.

The GPIO mask interrupt set registers (PORTxIO_MASKA_SET, PORTxIO_MASKB_SET) provide an alternative port to manipulate the GPIO mask interrupt registers. While a direct write to a mask interrupt register alters all bits in the register, writes to a mask interrupt set register can be used to set a single or a few bits only. No read-modify-write operations are required.

The mask interrupt set registers are write-1-to-set registers. All ones contained in the value written to the mask interrupt set register set the

respective bits in the mask interrupt register. The zeroes have no effect. Writing a one to any bit enables the interrupt for the respective GPIO.

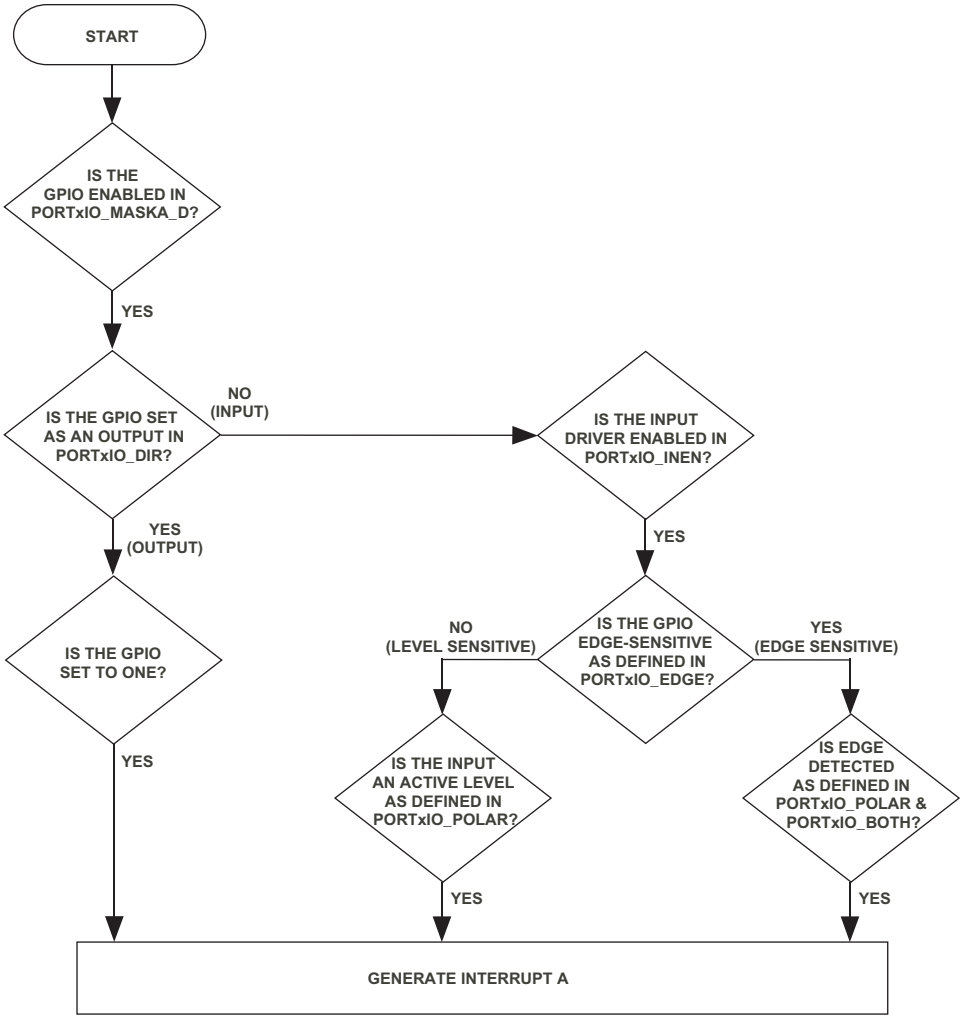

START
IS THE GPIO ENABLED IN PORTxIO_MASKA_D?
YES
IS THE GPIO SET AS AN OUTPUT IN PORTxIO_DIR?
NO (INPUT)
IS THE INPUT DRIVER ENABLED IN PORTxIO_INEN?
YES (OUTPUT)
YES
IS THE GPIO SET TO ONE?
NO (LEVEL SENSITIVE)
IS THE GPIO EDGE-SENSITIVE AS DEFINED IN PORTxIO_EDGE?
YES (EDGE SENSITIVE)
YES
IS THE INPUT AN ACTIVE LEVEL AS DEFINED IN PORTxIO_POLAR?
IS EDGE DETECTED AS DEFINED IN PORTxIO_POLAR & PORTxIO_BOTH?
YES
YES
GENERATE INTERRUPT A

Figure 9-1. GPIO Interrupt Generation Flow for Interrupt Channel A

The GPIO mask interrupt clear registers (PORTxIO_MASKA_CLEAR, PORTxIO_MASKB_CLEAR) provide an alternative port to manipulate the GPIO mask interrupt registers. While a direct write to a mask interrupt register alters all bits in the register, writes to the mask interrupt clear register can be used to clear a single bit or a few bits only. No read-modify-write operations are required.

The mask interrupt clear registers are write-1-to-clear registers. All ones contained in the value written to the mask interrupt clear register clear the respective bits in the mask interrupt register. The zeroes have no effect. Writing a one to any bit disables the interrupt for the respective GPIO.

The GPIO mask interrupt toggle registers (PORTxIO_MASKA_TOGGLE, PORTxIO_MASKB_TOGGLE) provide an alternative port to manipulate the GPIO mask interrupt registers. While a direct write to a mask interrupt register alters all bits in the register, writes to a mask interrupt toggle register can be used to toggle a single bit or a few bits only. No read-modify-write operations are required.

The mask interrupt toggle registers are write-1-to-clear registers. All ones contained in the value written to the mask interrupt toggle register toggle the respective bits in the mask interrupt register. The zeroes have no effect. Writing a one to any bit toggles the interrupt for the respective GPIO.

Figure 9-1 illustrates the interrupt flow of any GPIO module's interrupt A channel. The interrupt B channel behaves identically.

All GPIOs assigned to the same interrupt channel are OR'ed. If multiple GPIOs are assigned to the same interrupt channel, it is up to the interrupt service routine to evaluate the GPIO data registers to determine the signaling interrupt source.

Although each GPIO module provides two independent interrupt channels, the interrupt A channels of port F and port G are OR'ed as shown in

Figure 9-2. Therefore the total number of GPIO interrupt channels is five.

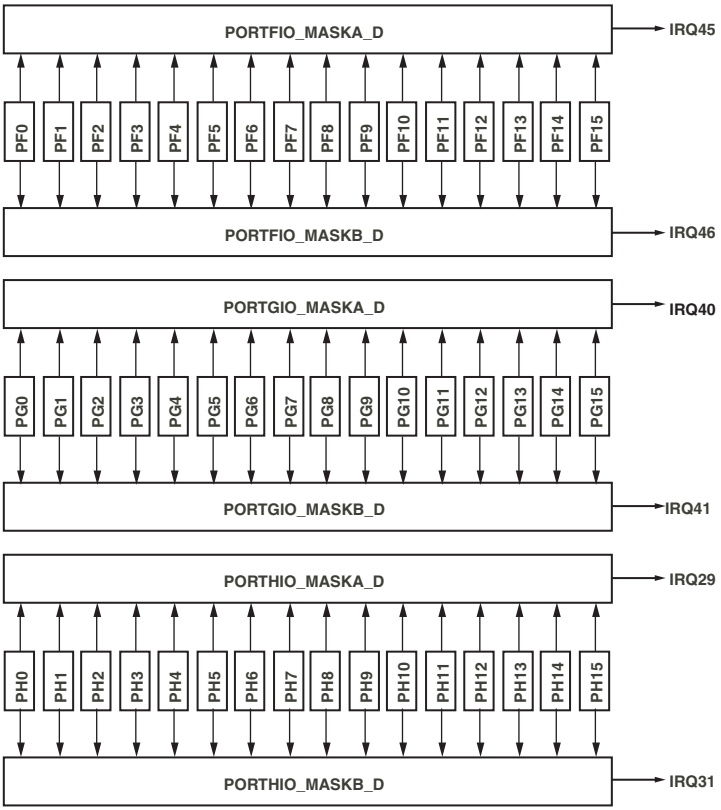


Figure 9-2. GPIO Interrupt Channels

Programming Model

Figure 9-3 and Figure 9-4 on page 9-23 show the programming model for the general-purpose ports.

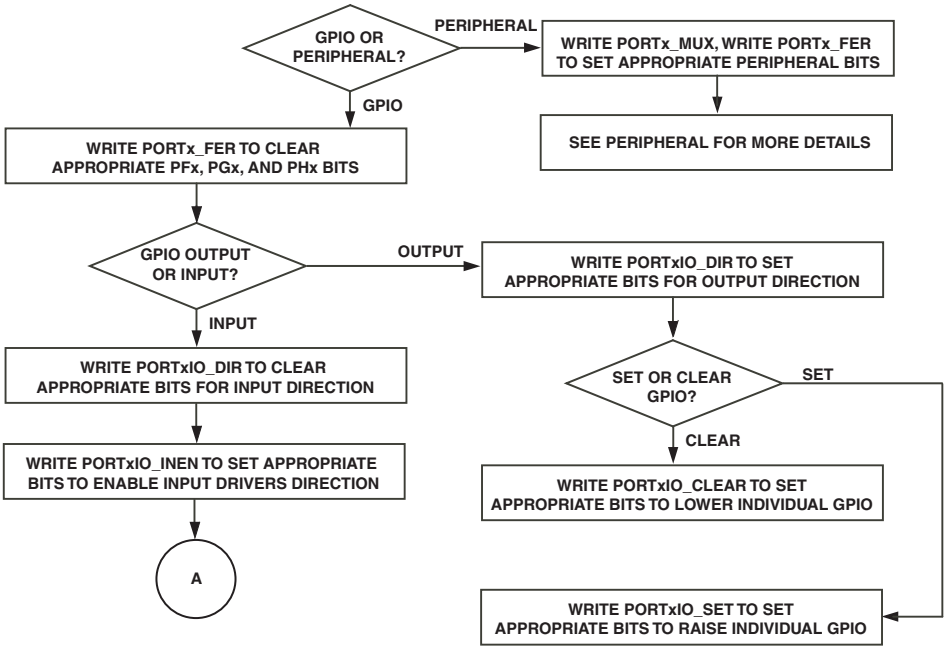

GPIO OR PERIPHERAL?
PERIPHERAL
WRITE PORTx_MUX, WRITE PORTx_FER TO SET APPROPRIATE PERIPHERAL BITS
SEE PERIPHERAL FOR MORE DETAILS
GPIO
WRITE PORTx_FER TO CLEAR APPROPRIATE PFx, PGx, AND PHx BITS
GPIO OUTPUT OR INPUT?
OUTPUT
WRITE PORTxIO_DIR TO SET APPROPRIATE BITS FOR OUTPUT DIRECTION
INPUT
WRITE PORTxIO_DIR TO CLEAR APPROPRIATE BITS FOR INPUT DIRECTION
SET OR CLEAR GPIO?
SET
CLEAR
WRITE PORTxIO_INEN TO SET APPROPRIATE BITS TO ENABLE INPUT DRIVERS DIRECTION
WRITE PORTxIO_CLEAR TO SET APPROPRIATE BITS TO LOWER INDIVIDUAL GPIO
A
WRITE PORTxIO_SET TO SET APPROPRIATE BITS TO RAISE INDIVIDUAL GPIO

Figure 9-3. GPIO Flow Chart (Part 1 of 2)

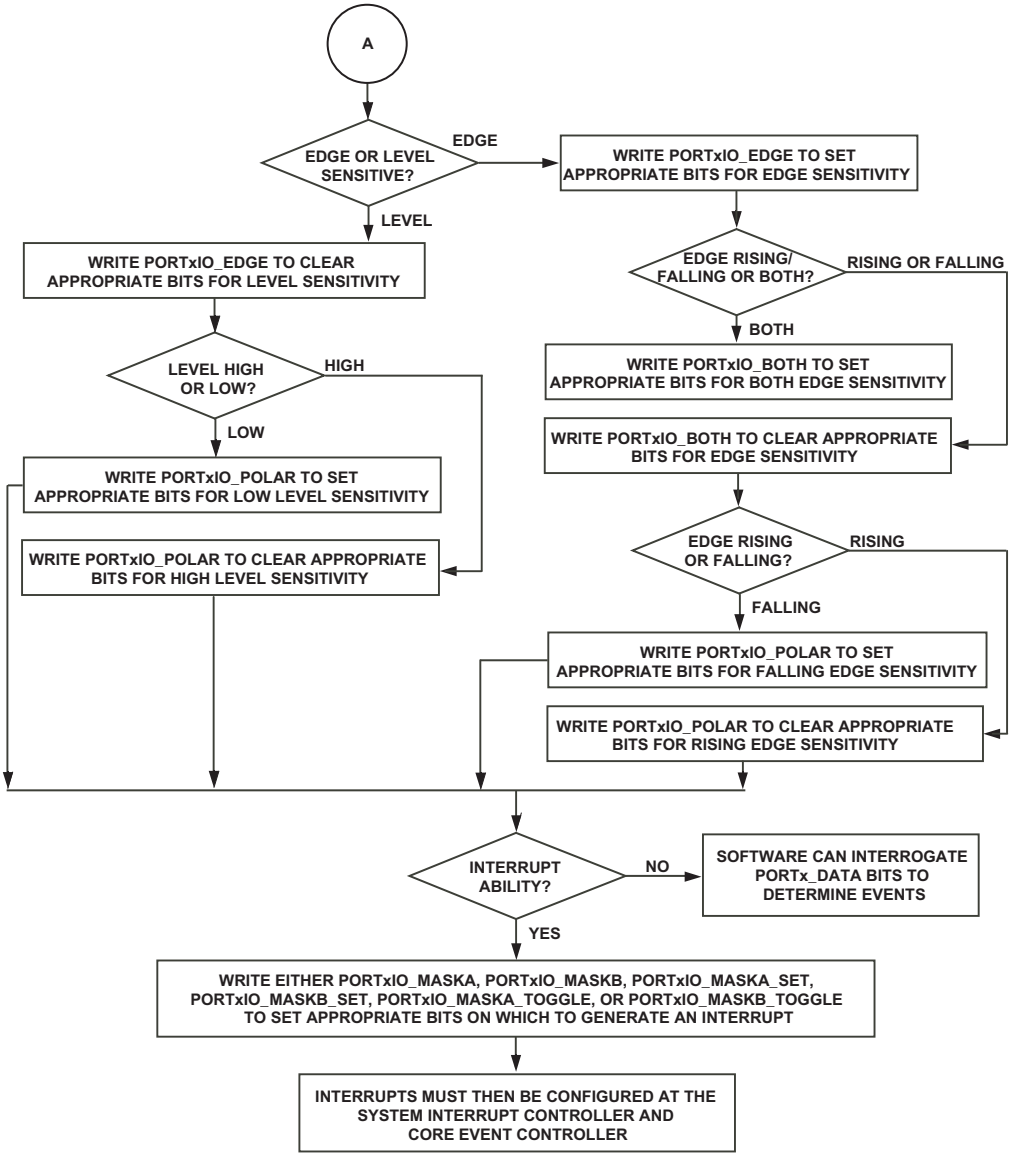

A
EDGE OR LEVEL SENSITIVE?
EDGE
LEVEL
WRITE PORTxIO_EDGE TO SET APPROPRIATE BITS FOR EDGE SENSITIVITY
WRITE PORTxIO_EDGE TO CLEAR APPROPRIATE BITS FOR LEVEL SENSITIVITY
EDGE RISING/ FALLING OR BOTH?
RISING OR FALLING
BOTH
LEVEL HIGH OR LOW?
HIGH
LOW
WRITE PORTxIO_BOTH TO SET APPROPRIATE BITS FOR BOTH EDGE SENSITIVITY
WRITE PORTxIO_BOTH TO CLEAR APPROPRIATE BITS FOR EDGE SENSITIVITY
WRITE PORTxIO_POLAR TO SET APPROPRIATE BITS FOR LOW LEVEL SENSITIVITY
WRITE PORTxIO_POLAR TO CLEAR APPROPRIATE BITS FOR HIGH LEVEL SENSITIVITY
EDGE RISING OR FALLING?
RISING
FALLING
WRITE PORTxIO_POLAR TO SET APPROPRIATE BITS FOR FALLING EDGE SENSITIVITY
WRITE PORTxIO_POLAR TO CLEAR APPROPRIATE BITS FOR RISING EDGE SENSITIVITY
INTERRUPT ABILITY?
NO
SOFTWARE CAN INTERROGATE PORTx_DATA BITS TO DETERMINE EVENTS
YES
WRITE EITHER PORTxIO_MASKA, PORTxIO_MASKB, PORTxIO_MASKA_SET, PORTxIO_MASKB_SET, PORTxIO_MASKA_TOGGLE, OR PORTxIO_MASKB_TOGGLE TO SET APPROPRIATE BITS ON WHICH TO GENERATE AN INTERRUPT
INTERRUPTS MUST THEN BE CONFIGURED AT THE SYSTEM INTERRUPT CONTROLLER AND CORE EVENT CONTROLLER

Figure 9-4. GPIO Flow Chart (Part 2 of 2)

GPIO Drive Hysteresis Control

The ADSP-BF52x contains additional registers that control input hysteresis for Port F, Port G and Port H. These are also included for pins other than GPIOs. Figure 9-5 on page 9-24 to Figure 9-9 on page 9-27 show the bit descriptions of these registers.

Portx Control (PORTx_HYSTERESIS) Register

This register configures Schmitt triggering (SE) for the PORTx inputs. The Schmitt trigger can be set only for pin groups, classified by the pin muxing controls. For each controlled group of pins, 00 implies disable and 01 implies enable Schmitt trigger. Combinations of 1x are reserved.

Port F Hysteresis Register (PORTF_HYSTERESIS)

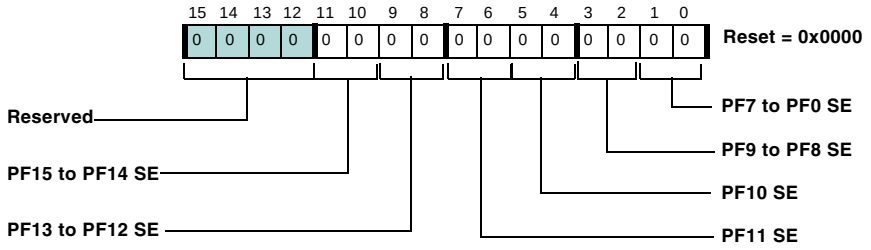


Figure 9-5. Port F Hysteresis Register

Port G Hysteresis Register (PORTG_HYSTERESIS)

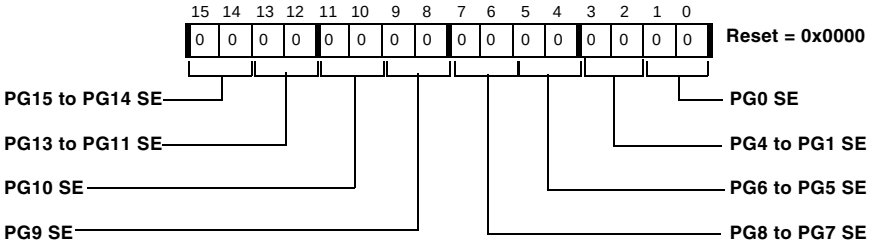


Figure 9-6. Port G Hysteresis Register

Port H Hysteresis Register (PORTH_HYSTERESIS)

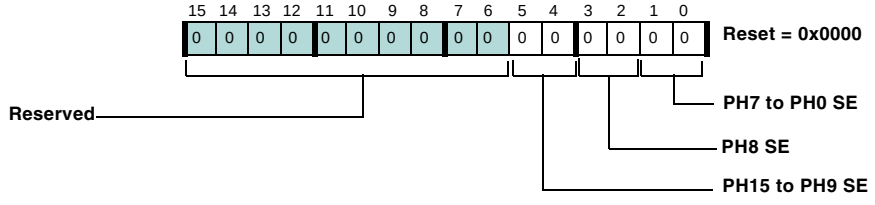


Figure 9-7. Port H Hysteresis Register

Hysteresis Control Register

This register sets the Schmitt trigger (SE) for various BF52x signals. For each controlled group of pins, 00 implies disable and 01 implies enable Schmitt trigger. Combinations of 1x are reserved.

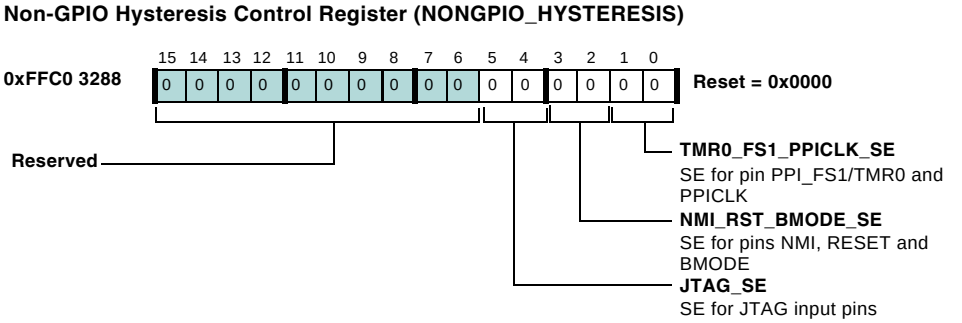

Non-GPIO Hysteresis Control Register (NONGPIO_HYSTERESIS)
0xFFC0 3288
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Reserved
TMR0_FS1_PPICLK_SE
SE for pin PPI_FS1/TMR0 and PPICLK
NMI_RST_BMODE_SE
SE for pins NMI, RESET and BMODE
JTAG_SE
SE for JTAG input pins

Figure 9-8. Non-GPIO Hysteresis Control Register

TWI Drive Strength Control Register

This register sets the drive strength and tolerance for the TWI signals on the ADSP-BF52x as specified in the diagram.

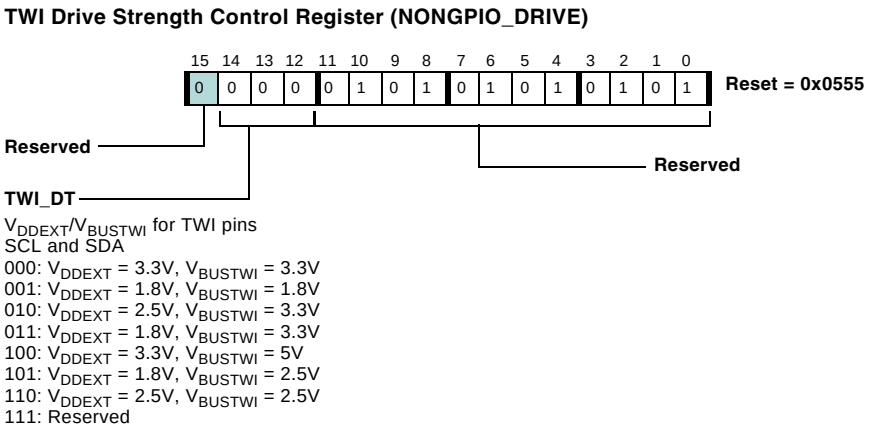

TWI Drive Strength Control Register (NONGPIO_DRIVE)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1
Reset = 0x0555
Reserved
Reserved
TWI_DT
VDDEXT/VBUSTWI for TWI pins SCL and SDA
000: VDDEXT = 3.3V, VBUSTWI = 3.3V
001: VDDEXT = 1.8V, VBUSTWI = 1.8V
010: VDDEXT = 2.5V, VBUSTWI = 3.3V
011: VDDEXT = 1.8V, VBUSTWI = 3.3V
100: VDDEXT = 3.3V, VBUSTWI = 5V
101: VDDEXT = 1.8V, VBUSTWI = 2.5V
110: VDDEXT = 2.5V, VBUSTWI = 2.5V
111: Reserved

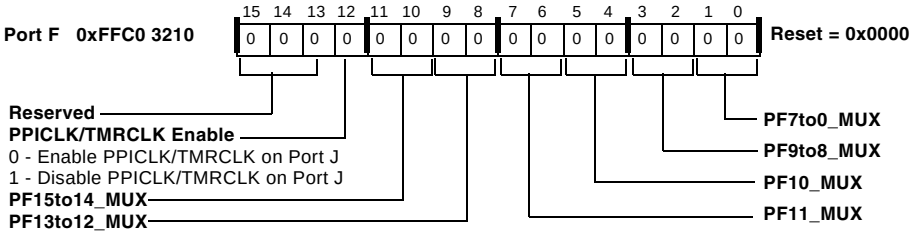
Figure 9-9. TWI Drive Strength Control Register

Memory-Mapped GPIO Registers

The GPIO registers are part of the system memory-mapped registers (MMRs). Figure 9-10 through Figure 9-30 on page 9-42 illustrate the GPIO registers. The addresses of the programmable flag MMRs appear in Appendix A, “System MMR Assignments”.

Port Multiplexer Control Register (PORTx_MUX)

Port F Multiplexer Control Register (PORTF_MUX)

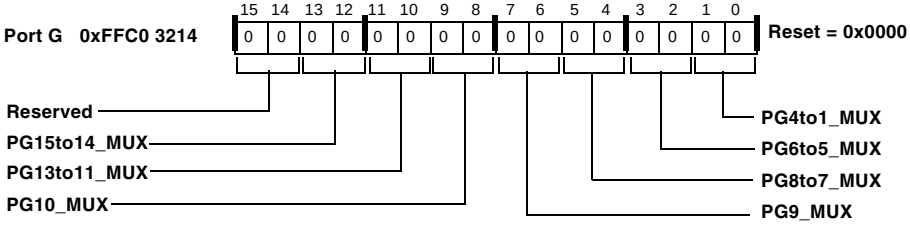

Port F 0xFFC0 3210
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Reserved
PPICLK/TMRCLK Enable
0 - Enable PPICLK/TMRCLK on Port J
1 - Disable PPICLK/TMRCLK on Port J
PF15to14_MUX
PF13to12_MUX
PF7to0_MUX
PF9to8_MUX
PF10_MUX
PF11_MUX

For all bit fields:
00 = first peripheral function
01 = first alternate peripheral function
10 = second alternate peripheral function
11 = Reserved

Refer to Table 9-2 on page 9-5 to Table 9-4 on page 9-8 for reserved bits in the PORTF_MUX register.

Figure 9-10. Port F Multiplexer Control Register

Port G Multiplexer Control Register (PORTG_MUX)

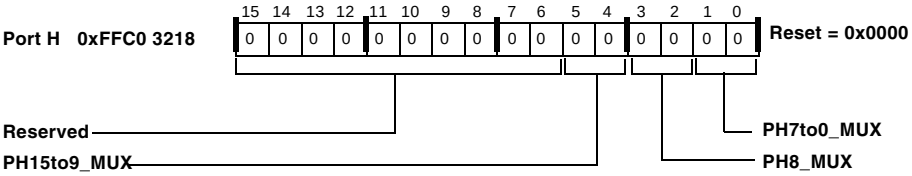


For all bit fields:
00 = first peripheral function
01 = first alternate peripheral function
10 = second alternate peripheral function
11 = Reserved

Refer to Table 9-2 on page 9-5 to Table 9-4 on page 9-8 for reserved bits in the PORTG_MUX register.

Figure 9-11. Port G Multiplexer Control Register

Port H Multiplexer Control Register (PORTH_MUX)



For all bit fields:
- 00 = first peripheral function
- 01 = first alternate peripheral function
- 10 = second alternate peripheral function
- 11 = Reserved

Refer to Table 9-2 on page 9-5 to Table 9-4 on page 9-8 for reserved bits in the PORTH_MUX register.

Figure 9-12. Port H Multiplexer Control Register

Function Enable Registers (PORTx_FER)

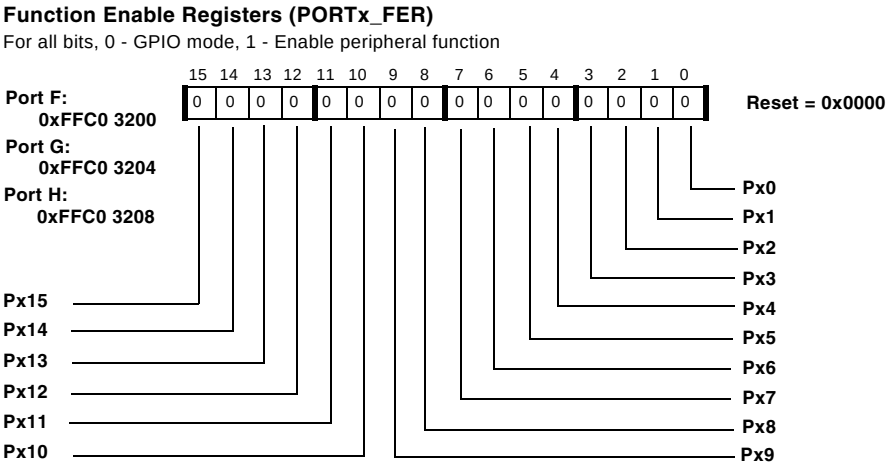


Figure 9-13. Function Enable Registers

GPIO Direction Registers (PORTxIO_DIR)

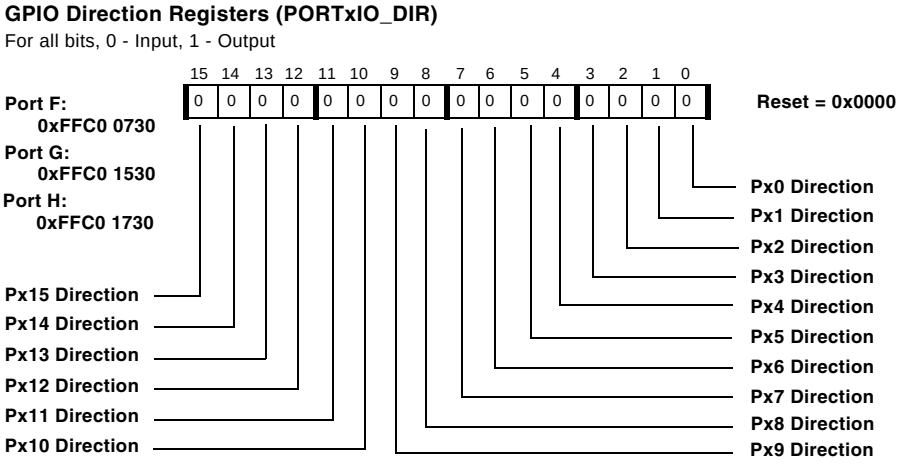


Figure 9-14. GPIO Direction Registers

GPIO Input Enable Registers (PORTxIO_INEN)

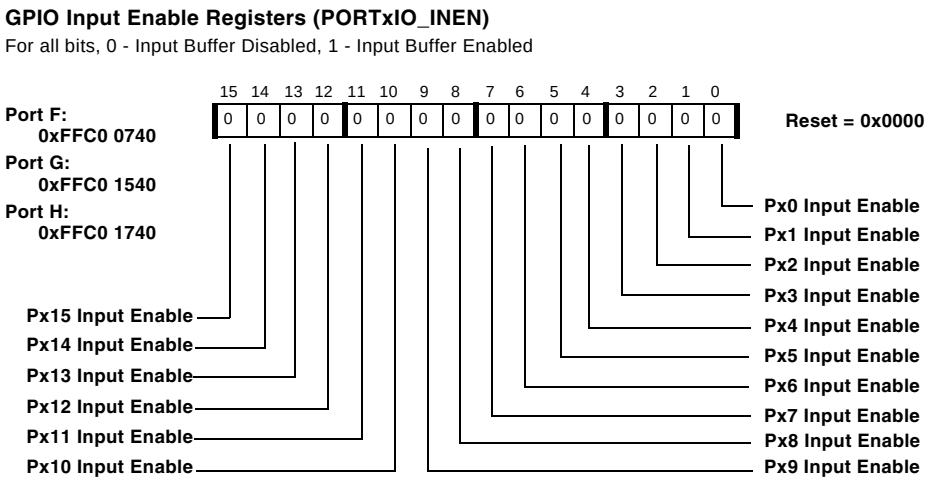

GPIO Input Enable Registers (PORTxIO_INEN)
For all bits, 0 - Input Buffer Disabled, 1 - Input Buffer Enabled
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Port F:
0xFFC0 0740
Port G:
0xFFC0 1540
Port H:
0xFFC0 1740
Reset = 0x0000
Px0 Input Enable
Px1 Input Enable
Px2 Input Enable
Px3 Input Enable
Px4 Input Enable
Px5 Input Enable
Px6 Input Enable
Px7 Input Enable
Px8 Input Enable
Px9 Input Enable
Px10 Input Enable
Px11 Input Enable
Px12 Input Enable
Px13 Input Enable
Px14 Input Enable
Px15 Input Enable

Figure 9-15. GPIO Input Enable Registers

GPIO Data Registers (PORTxIO)

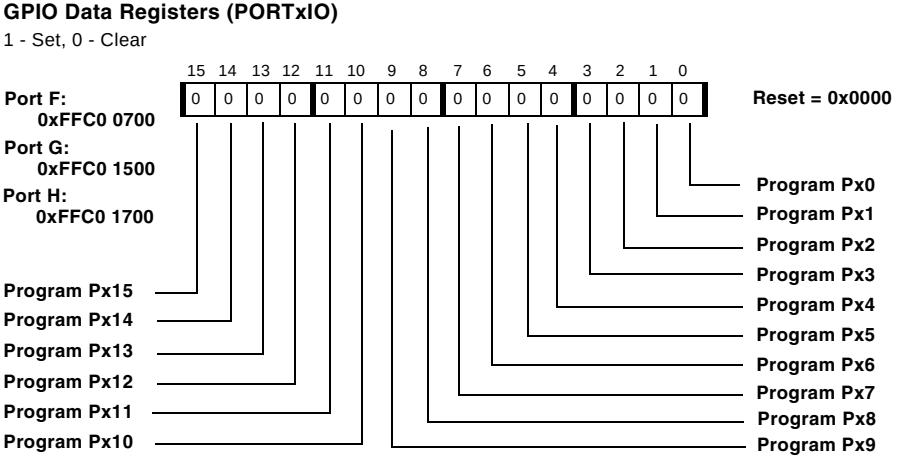


Figure 9-16. GPIO Data Registers

GPIO Set Registers (PORTxIO_SET)

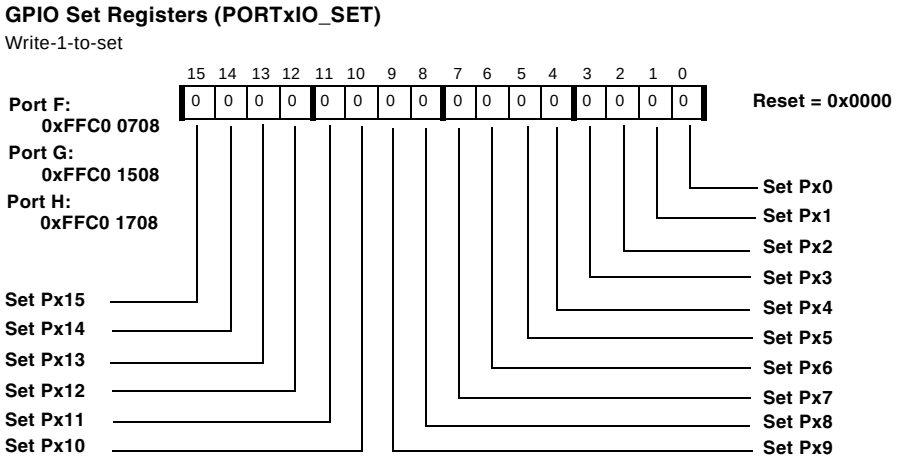


Figure 9-17. GPIO Set Registers

GPIO Clear Registers (PORTxIO_CLEAR)

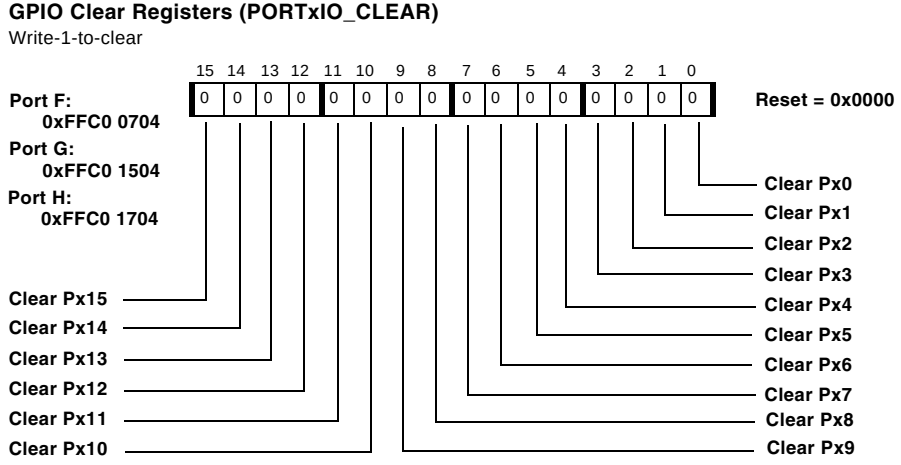


Figure 9-18. GPIO Clear Registers

GPIO Toggle Registers (PORTxIO_TOGGLE)

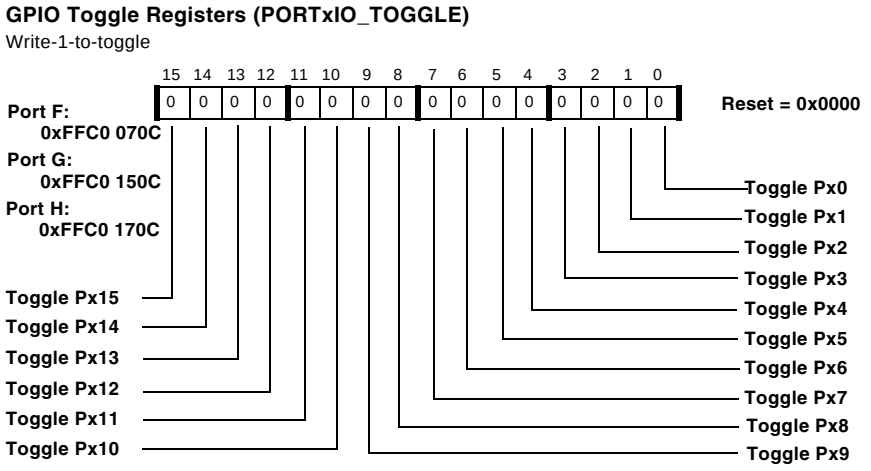


Figure 9-19. GPIO Toggle Registers

GPIO Polarity Registers (PORTxIO_POLAR)

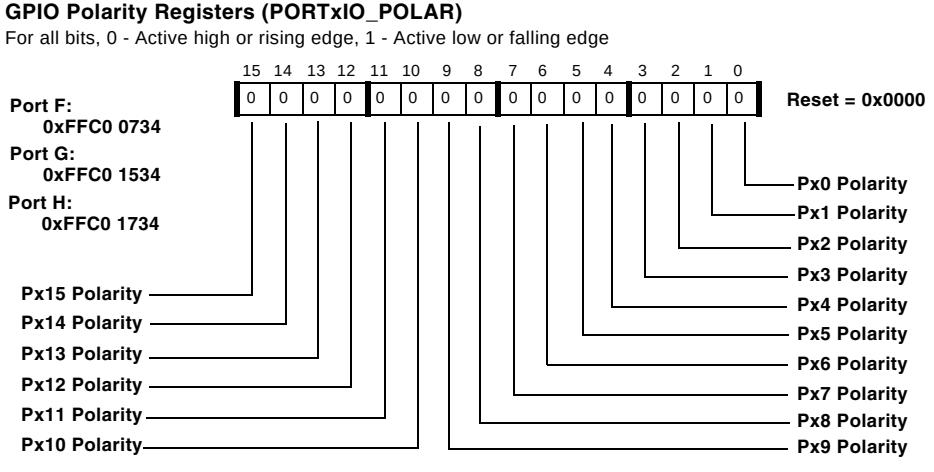


Figure 9-20. GPIO Polarity Registers

Interrupt Sensitivity Registers (PORTxIO_EDGE)

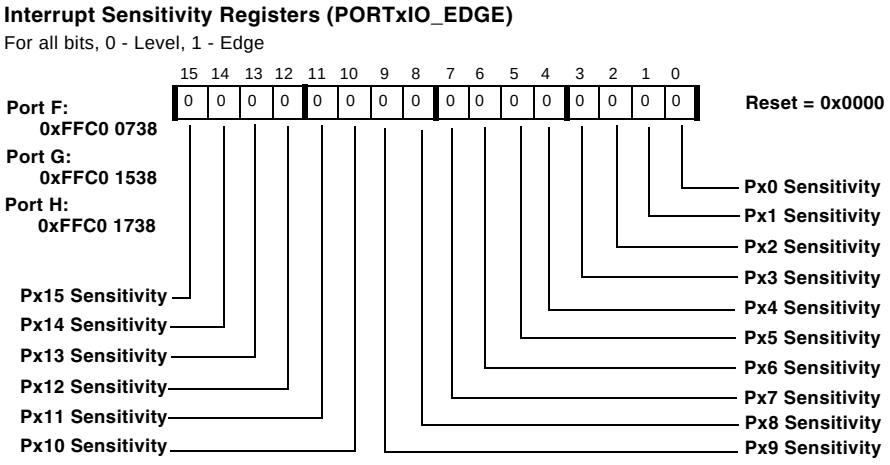


Figure 9-21. Interrupt Sensitivity Registers

GPIO Set on Both Edges Registers (PORTxIO_BOTH)

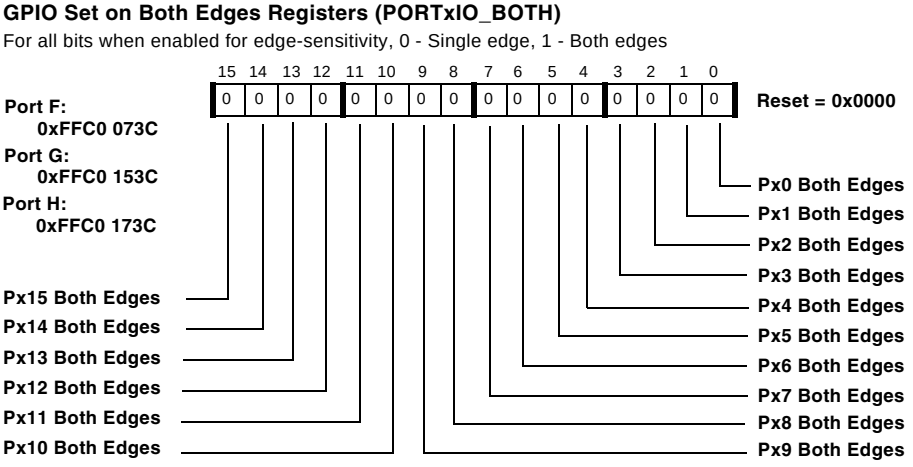

GPIO Set on Both Edges Registers (PORTxIO_BOTH)
For all bits when enabled for edge-sensitivity, 0 - Single edge, 1 - Both edges
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Port F:
0xFFC0 073C
Port G:
0xFFC0 153C
Port H:
0xFFC0 173C
Px15 Both Edges
Px14 Both Edges
Px13 Both Edges
Px12 Both Edges
Px11 Both Edges
Px10 Both Edges
Px0 Both Edges
Px1 Both Edges
Px2 Both Edges
Px3 Both Edges
Px4 Both Edges
Px5 Both Edges
Px6 Both Edges
Px7 Both Edges
Px8 Both Edges
Px9 Both Edges

Figure 9-22. GPIO Set on Both Edges Registers

GPIO Mask Interrupt Registers (PORTxIO_MASKA/B)

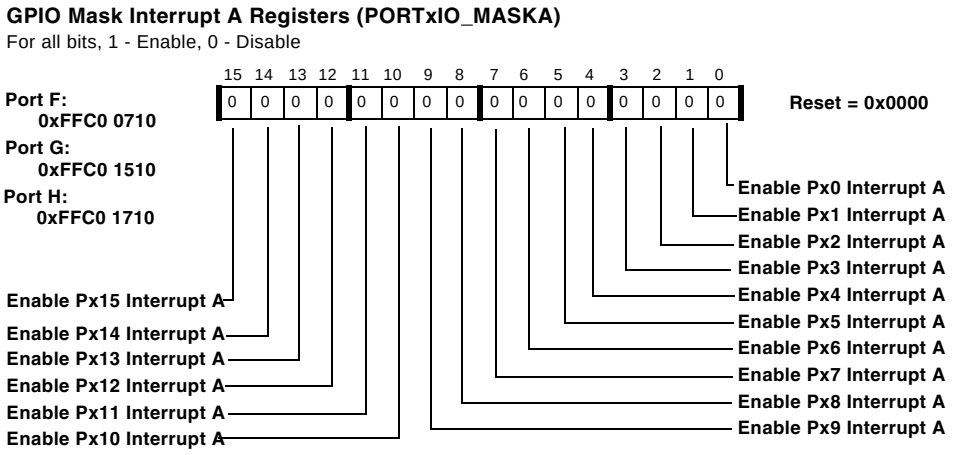

GPIO Mask Interrupt A Registers (PORTxIO_MASKA)
For all bits, 1 - Enable, 0 - Disable
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Port F:
0xFFC0 0710
Port G:
0xFFC0 1510
Port H:
0xFFC0 1710
Enable Px15 Interrupt A
Enable Px14 Interrupt A
Enable Px13 Interrupt A
Enable Px12 Interrupt A
Enable Px11 Interrupt A
Enable Px10 Interrupt A
Enable Px0 Interrupt A
Enable Px1 Interrupt A
Enable Px2 Interrupt A
Enable Px3 Interrupt A
Enable Px4 Interrupt A
Enable Px5 Interrupt A
Enable Px6 Interrupt A
Enable Px7 Interrupt A
Enable Px8 Interrupt A
Enable Px9 Interrupt A

Figure 9-23. GPIO Mask Interrupt A Registers

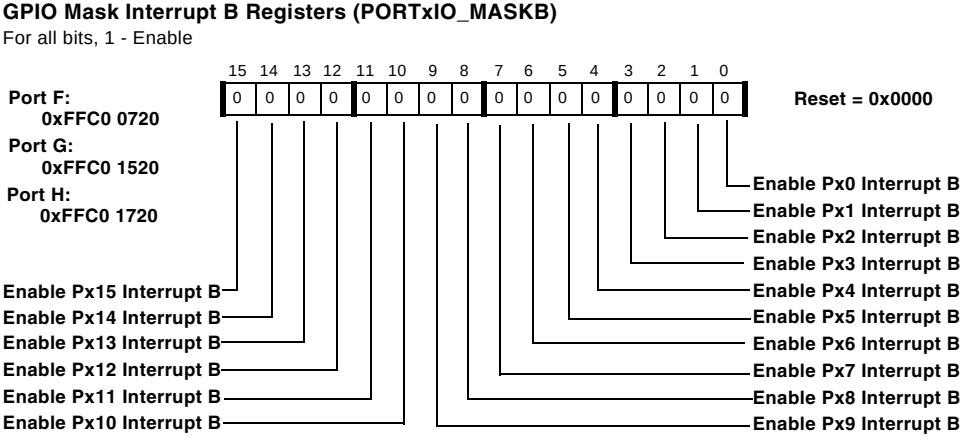

GPIO Mask Interrupt B Registers (PORTxIO_MASKB)
For all bits, 1 - Enable
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Port F:
0xFFC0 0720
Port G:
0xFFC0 1520
Port H:
0xFFC0 1720
Enable Px15 Interrupt B
Enable Px14 Interrupt B
Enable Px13 Interrupt B
Enable Px12 Interrupt B
Enable Px11 Interrupt B
Enable Px10 Interrupt B
Enable Px0 Interrupt B
Enable Px1 Interrupt B
Enable Px2 Interrupt B
Enable Px3 Interrupt B
Enable Px4 Interrupt B
Enable Px5 Interrupt B
Enable Px6 Interrupt B
Enable Px7 Interrupt B
Enable Px8 Interrupt B
Enable Px9 Interrupt B

Figure 9-24. GPIO Mask Interrupt B Registers

GPIO Mask Interrupt Set Registers (PORTxIO_MASKA/B_SET)

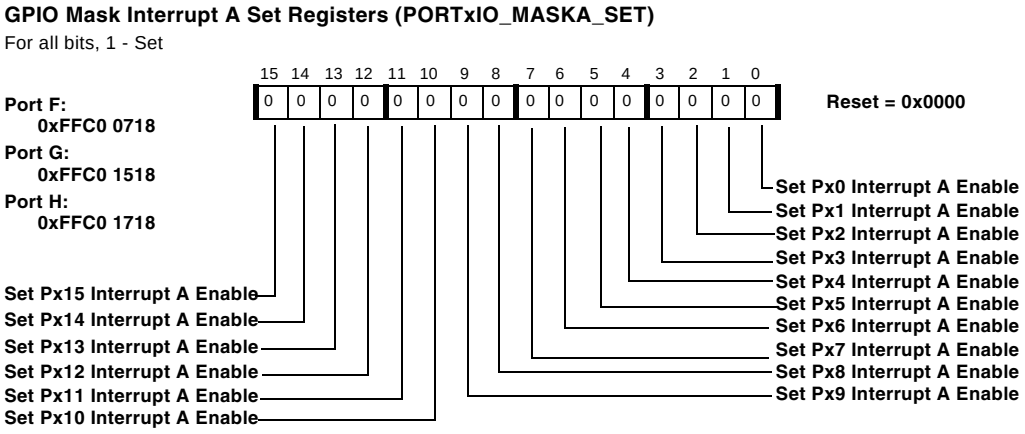

GPIO Mask Interrupt A Set Registers (PORTxIO_MASKA_SET)
For all bits, 1 - Set
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Port F:
0xFFC0 0718
Port G:
0xFFC0 1518
Port H:
0xFFC0 1718
Set Px15 Interrupt A Enable
Set Px14 Interrupt A Enable
Set Px13 Interrupt A Enable
Set Px12 Interrupt A Enable
Set Px11 Interrupt A Enable
Set Px10 Interrupt A Enable
Set Px9 Interrupt A Enable
Set Px8 Interrupt A Enable
Set Px7 Interrupt A Enable
Set Px6 Interrupt A Enable
Set Px5 Interrupt A Enable
Set Px4 Interrupt A Enable
Set Px3 Interrupt A Enable
Set Px2 Interrupt A Enable
Set Px1 Interrupt A Enable
Set Px0 Interrupt A Enable

Figure 9-25. GPIO Mask Interrupt A Set Registers

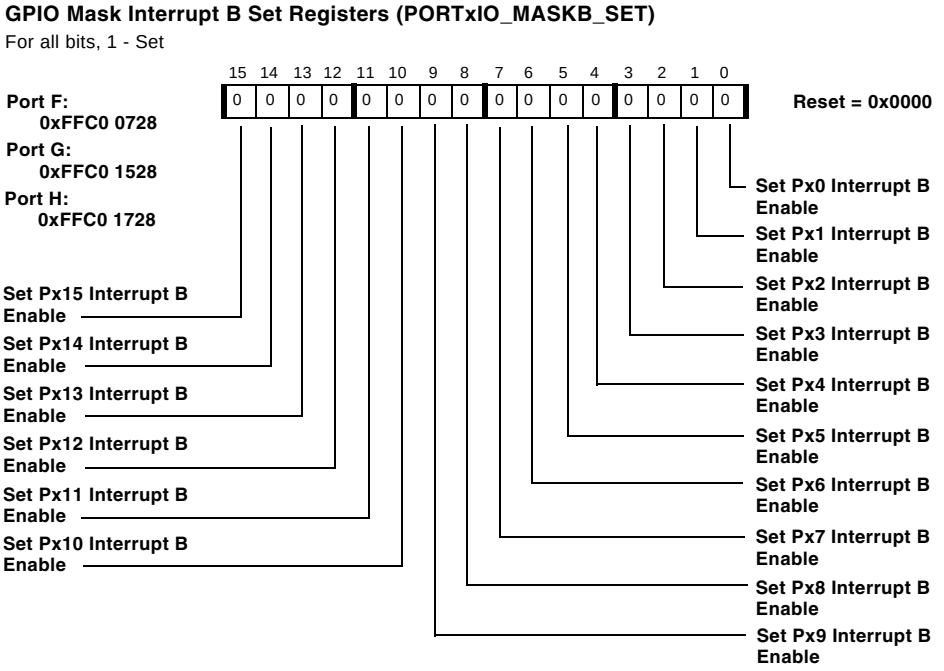

GPIO Mask Interrupt B Set Registers (PORTxIO_MASKB_SET)
For all bits, 1 - Set
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Port F:
0xFFC0 0728
Port G:
0xFFC0 1528
Port H:
0xFFC0 1728
Set Px15 Interrupt B Enable
Set Px14 Interrupt B Enable
Set Px13 Interrupt B Enable
Set Px12 Interrupt B Enable
Set Px11 Interrupt B Enable
Set Px10 Interrupt B Enable
Set Px0 Interrupt B Enable
Set Px1 Interrupt B Enable
Set Px2 Interrupt B Enable
Set Px3 Interrupt B Enable
Set Px4 Interrupt B Enable
Set Px5 Interrupt B Enable
Set Px6 Interrupt B Enable
Set Px7 Interrupt B Enable
Set Px8 Interrupt B Enable
Set Px9 Interrupt B Enable

Figure 9-26. GPIO Mask Interrupt B Set Registers

GPIO Mask Interrupt Clear Registers (PORTxIO_MASKA/B_CLEAR)

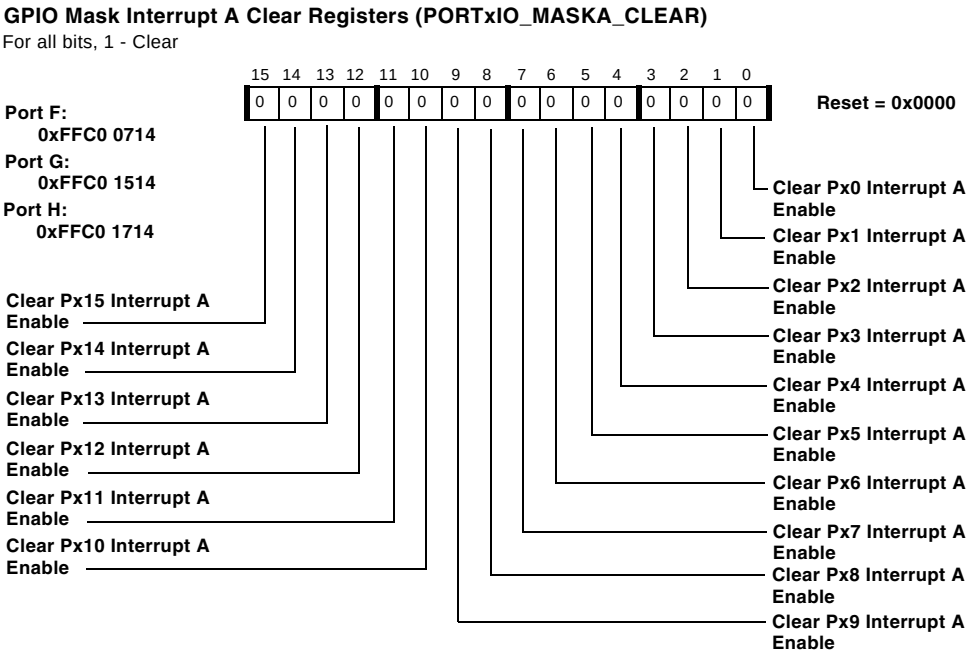

GPIO Mask Interrupt A Clear Registers (PORTxIO_MASKA_CLEAR)
For all bits, 1 - Clear
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Port F:
0xFFC0 0714
Port G:
0xFFC0 1514
Port H:
0xFFC0 1714
Clear Px15 Interrupt A Enable
Clear Px14 Interrupt A Enable
Clear Px13 Interrupt A Enable
Clear Px12 Interrupt A Enable
Clear Px11 Interrupt A Enable
Clear Px10 Interrupt A Enable
Clear Px0 Interrupt A Enable
Clear Px1 Interrupt A Enable
Clear Px2 Interrupt A Enable
Clear Px3 Interrupt A Enable
Clear Px4 Interrupt A Enable
Clear Px5 Interrupt A Enable
Clear Px6 Interrupt A Enable
Clear Px7 Interrupt A Enable
Clear Px8 Interrupt A Enable
Clear Px9 Interrupt A Enable

Figure 9-27. GPIO Mask Interrupt A Clear Registers

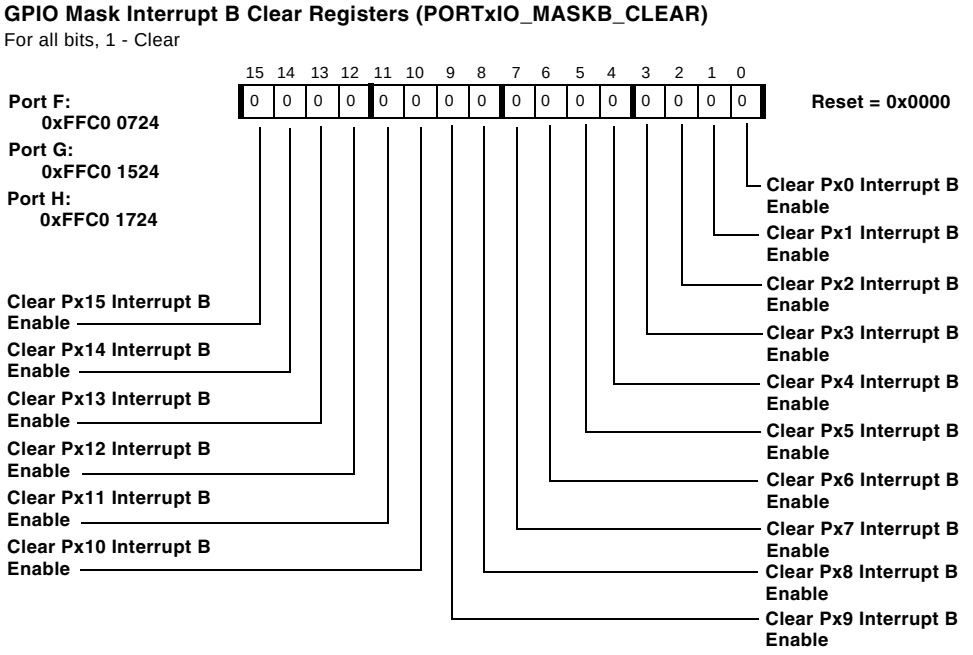

GPIO Mask Interrupt B Clear Registers (PORTxIO_MASKB_CLEAR)
For all bits, 1 - Clear
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Port F:
0xFFC0 0724
Port G:
0xFFC0 1524
Port H:
0xFFC0 1724
Clear Px15 Interrupt B Enable
Clear Px14 Interrupt B Enable
Clear Px13 Interrupt B Enable
Clear Px12 Interrupt B Enable
Clear Px11 Interrupt B Enable
Clear Px10 Interrupt B Enable
Clear Px0 Interrupt B Enable
Clear Px1 Interrupt B Enable
Clear Px2 Interrupt B Enable
Clear Px3 Interrupt B Enable
Clear Px4 Interrupt B Enable
Clear Px5 Interrupt B Enable
Clear Px6 Interrupt B Enable
Clear Px7 Interrupt B Enable
Clear Px8 Interrupt B Enable
Clear Px9 Interrupt B Enable

Figure 9-28. GPIO Mask Interrupt B Clear Registers

GPIO Mask Interrupt Toggle Registers (PORTxIO_MASKA/B_TOGGLE)

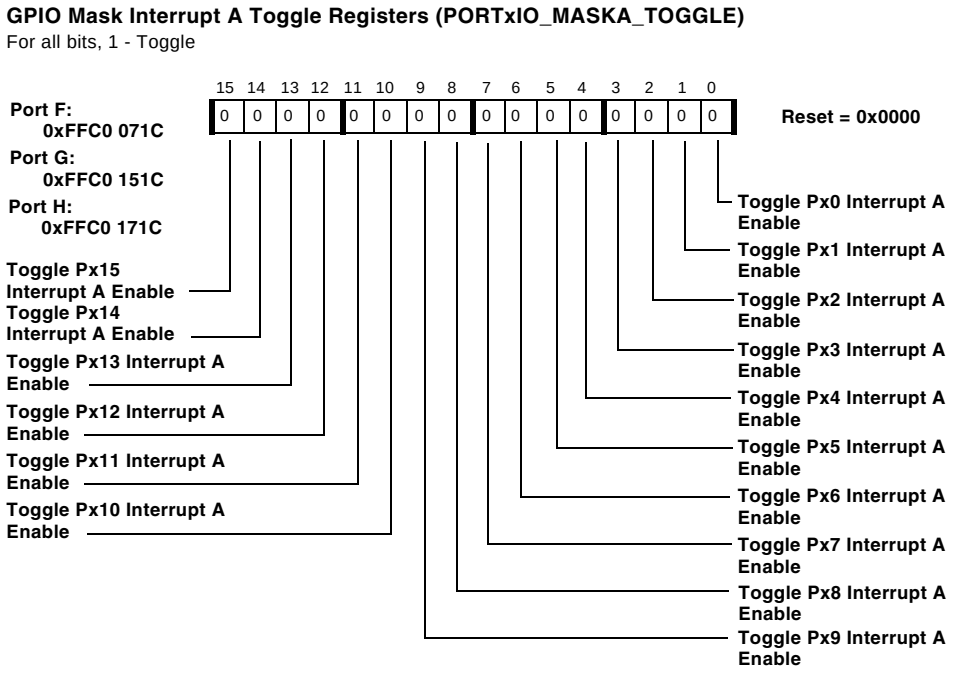

GPIO Mask Interrupt A Toggle Registers (PORTxIO_MASKA_TOGGLE)
For all bits, 1 - Toggle
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Port F:
0xFFC0 071C
Port G:
0xFFC0 151C
Port H:
0xFFC0 171C
Toggle Px15 Interrupt A Enable
Toggle Px14 Interrupt A Enable
Toggle Px13 Interrupt A Enable
Toggle Px12 Interrupt A Enable
Toggle Px11 Interrupt A Enable
Toggle Px10 Interrupt A Enable
Toggle Px0 Interrupt A Enable
Toggle Px1 Interrupt A Enable
Toggle Px2 Interrupt A Enable
Toggle Px3 Interrupt A Enable
Toggle Px4 Interrupt A Enable
Toggle Px5 Interrupt A Enable
Toggle Px6 Interrupt A Enable
Toggle Px7 Interrupt A Enable
Toggle Px8 Interrupt A Enable
Toggle Px9 Interrupt A Enable

Figure 9-29. GPIO Mask Interrupt A Toggle Registers

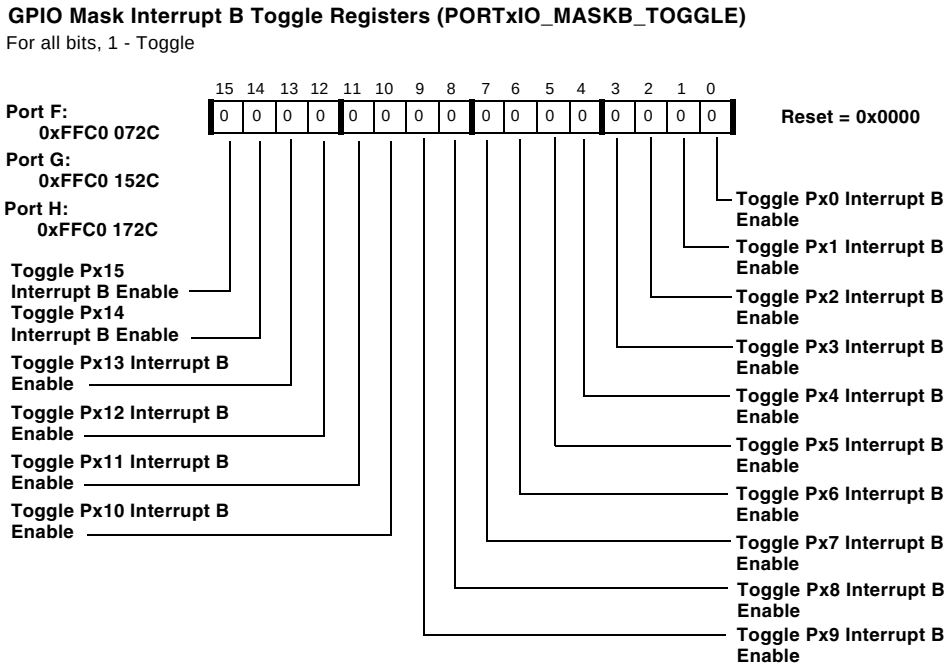

GPIO Mask Interrupt B Toggle Registers (PORTxIO_MASKB_TOGGLE)
For all bits, 1 - Toggle
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Port F:
0xFFC0 072C
Port G:
0xFFC0 152C
Port H:
0xFFC0 172C
Toggle Px15 Interrupt B Enable
Toggle Px14 Interrupt B Enable
Toggle Px13 Interrupt B Enable
Toggle Px12 Interrupt B Enable
Toggle Px11 Interrupt B Enable
Toggle Px10 Interrupt B Enable
Toggle Px0 Interrupt B Enable
Toggle Px1 Interrupt B Enable
Toggle Px2 Interrupt B Enable
Toggle Px3 Interrupt B Enable
Toggle Px4 Interrupt B Enable
Toggle Px5 Interrupt B Enable
Toggle Px6 Interrupt B Enable
Toggle Px7 Interrupt B Enable
Toggle Px8 Interrupt B Enable
Toggle Px9 Interrupt B Enable

Figure 9-30. GPIO Mask Interrupt B Toggle Registers

Programming Examples

Listing 9-1 provides examples for using the general-purpose ports.

Listing 9-1. General-Purpose Ports

```
/* set port f function enable register to GPIO (not peripheral)
*/
p0.l = lo(PORTF_FER);
p0.h = hi(PORTF_FER);

R0.h = 0x0000;
r0.l = 0x0000;
```

```
w[p0] = r0;

/* set port f direction register to enable some GPIO as output,
remaining are input */
p0.l = lo(PORTFIO_DIR);
p0.h = hi(PORTFIO_DIR);
r0.h = 0x0000;
r0.l = 0x0FC0;
w[p0] = r0;
ssync;

/* set port f clear register */
p0.l = lo(PORTFIO_CLEAR);
p0.h = hi(PORTFIO_CLEAR);
    r0.l = 0xFC0;
    w[p0] = r0;
    ssync;

/* set port f input enable register to enable input drivers of
some GPIOs */
p0.l = lo(PORTFIO_INEN);
p0.h = hi(PORTFIO_INEN);
r0.h = 0x0000;
r0.l = 0x003C;
w[p0] = r0;
ssync;

/* set port f polarity register */
p0.l = lo(PORTFIO_POLAR);
p0.h = hi(PORTFIO_POLAR);
r0 = 0x00000;
w[p0] = r0;
ssync;
```

10 GENERAL-PURPOSE TIMERS

This chapter describes the general-purpose (GP) timer module. Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF52x

For details regarding the number of GP timers for the ADSP-BF52x product, please refer to the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet*.

For GP Timer interrupt vector assignments, refer to Table 5-3 on page 5-18 in Chapter 5, "System Interrupts".

To determine how each of the GP Timers is multiplexed with other functional pins, refer to Table 9-2 on page 9-5 through Table 9-5 on page 9-9 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for each GP Timer, refer to Appendix A, "System MMR Assignments".

GP timer behavior for the ADSP-BF52x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Behavior for the ADSP-BF52x Processor" on page 10-57

Overview

The general-purpose timers support the following operating modes:

- Single-shot mode for interval timing and single pulse generation
- Pulse width modulation (PWM) generation with consistent update of period and pulse width values
- External signal capture mode with consistent update of period and pulse width values
- External event counter mode

Feature highlights are:

- Synchronous operation
- Consistent management of period and pulse width values
- Interaction with PPI module for video frame sync operation
- Autobaud detection for UART module
- Graceful bit pattern termination when stopping
- Support for center-aligned PWM patterns
- Error detection on implausible pattern values
- All read and write accesses to 32-bit registers are atomic
- Every timer has its dedicated interrupt request output
- Unused timers can function as edge-sensitive pin interrupts

The internal structure of the individual timers is illustrated by Figure 10-1, which shows the details of timer 0 as a representative example. The other timers have identical structure.

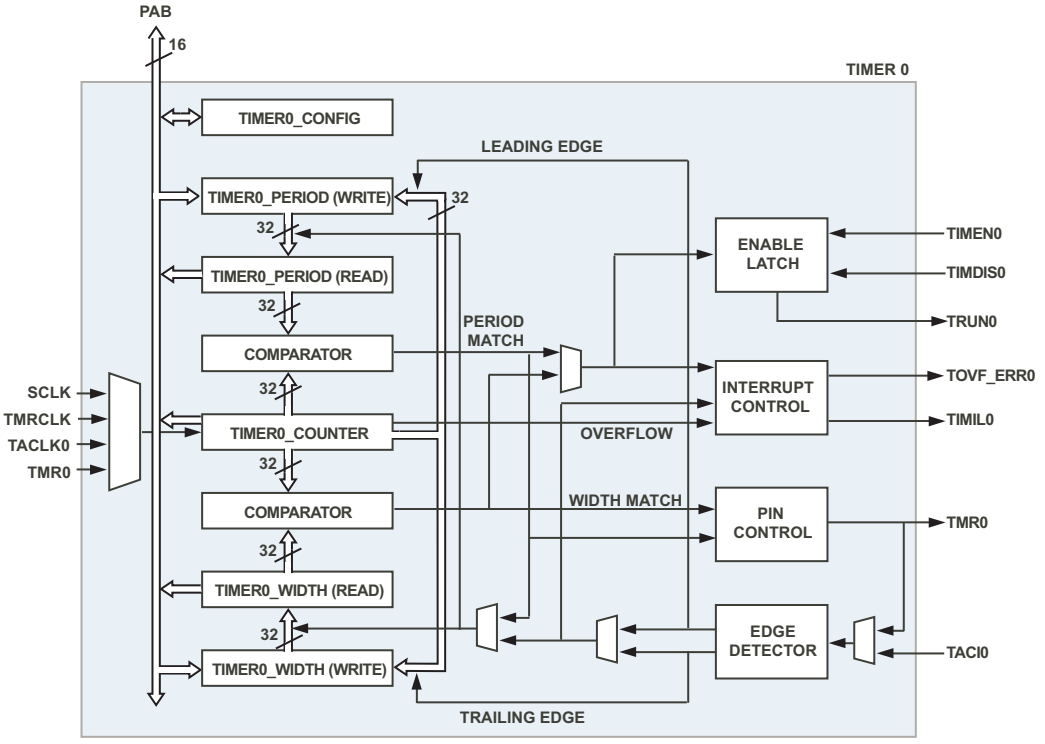


Figure 10-1. Internal Timer Structure

External Interface

Every timer has a dedicated TMR pin. If enabled, the TMR pins output the single-pulse or PWM signals generated by the timer. The TMR pins function as input in capture and counter modes. Polarity of the signals is programmable.

When clocked internally, the clock source is the processor's peripheral clock (SCLK). Assuming the peripheral clock is running at 133 MHz, the maximum period for the timer count is $((2^{32}-1) / 133\ \text{MHz}) = 32.2$ seconds.

Clock and capture input pins are sampled every SCLK cycle. The duration of every low or high state must be at least one SCLK. Therefore, the maximum allowed frequency of timer input signals is SCLK/2.

Internal Interface

Timer registers are always accessed by the core through the 16-bit PAB bus. Hardware ensures that all read and write operations from and to 32-bit timer registers are atomic.

Every timer has a dedicated interrupt request output that connects to the system interrupt controller (SIC).

Description of Operation

The core of every timer is a 32-bit counter, that can be interrogated through the read-only TIMER_COUNTER register. Depending on the mode of operation, the counter is reset to either 0x0000 0000 or 0x0000 0001 when the timer is enabled. The counter always counts upward. Usually, it is clocked by SCLK. In PWM mode it can be clocked by the alternate clock input TACLK or, alternatively, the common timer clock input TMRCLK. In counter mode, the counter is clocked by edges on the TMR input pin. The significant edge is programmable.

After 2^{32}-1 clocks, the counter overflows. This is reported by the overflow/error bit TOVF_ERR in the TIMER_STATUS register. In PWM and counter mode, the counter is reset by hardware when its content reaches the values stored in the TIMER_PERIOD register. In capture mode, the counter is reset by leading edges on the TMR or TACI input pin. If enabled, these events cause the interrupt latch TIMIL in the TIMER_STATUS register to be set and issue a system interrupt request. The TOVF_ERR and TIMIL latches are sticky and should be cleared by software using W1C (write-1-to-clear) operations to clear the interrupt request. The global

TIMER_STATUS register is 32-bits wide. A single atomic 32-bit read can report the status of all corresponding timers.

Before a timer can be enabled, its mode of operation is programmed in the individual timer-specific TIMER_CONFIG register. Then, the timers are started by writing a "1" to the representative bits in the global TIMER_ENABLE register.

The TIMER_ENABLE register can be used to enable all timers simultaneously. The register contains W1S (write-1-to-set) control bits, one for each timer. Correspondingly, the TIMER_DISABLE register contains W1C control bits to allow simultaneous or independent disabling of the timers. Either register can be read to check the enable status of the timers. A "1" indicates that the corresponding timer is enabled. The timer starts counting three SCLK cycles after the TIMEN bit is set.

While the PWM mode is used to generate PWM patterns, the capture mode (WDTH_CAP) is designed to “receive” PWM signals. A PWM pattern is represented by a pulse width and a signal period. This is described by the TIMER_WIDTH and TIMER_PERIOD register pair. In capture mode these registers are read only. Hardware always captures both values. Regardless of whether in PWM or capture mode, shadow buffers always ensure consistency between the TIMER_WIDTH and TIMER_PERIOD values. In PWM mode, hardware performs a plausibility check by the time the timer is enabled. If there is an error, the type is reported by the TIMER_CONFIG register and signalled by the TOVF_ERR bit.

Interrupt Processing

Each timer can generate a single interrupt. The resulting interrupt signals are routed to the system interrupt controller block for prioritization and masking. The timer status (TIMER_STATUS) register latches the timer interrupts to provide a means for software to determine the interrupt source.

Figure 10-2 shows the interrupt structure of the timers.

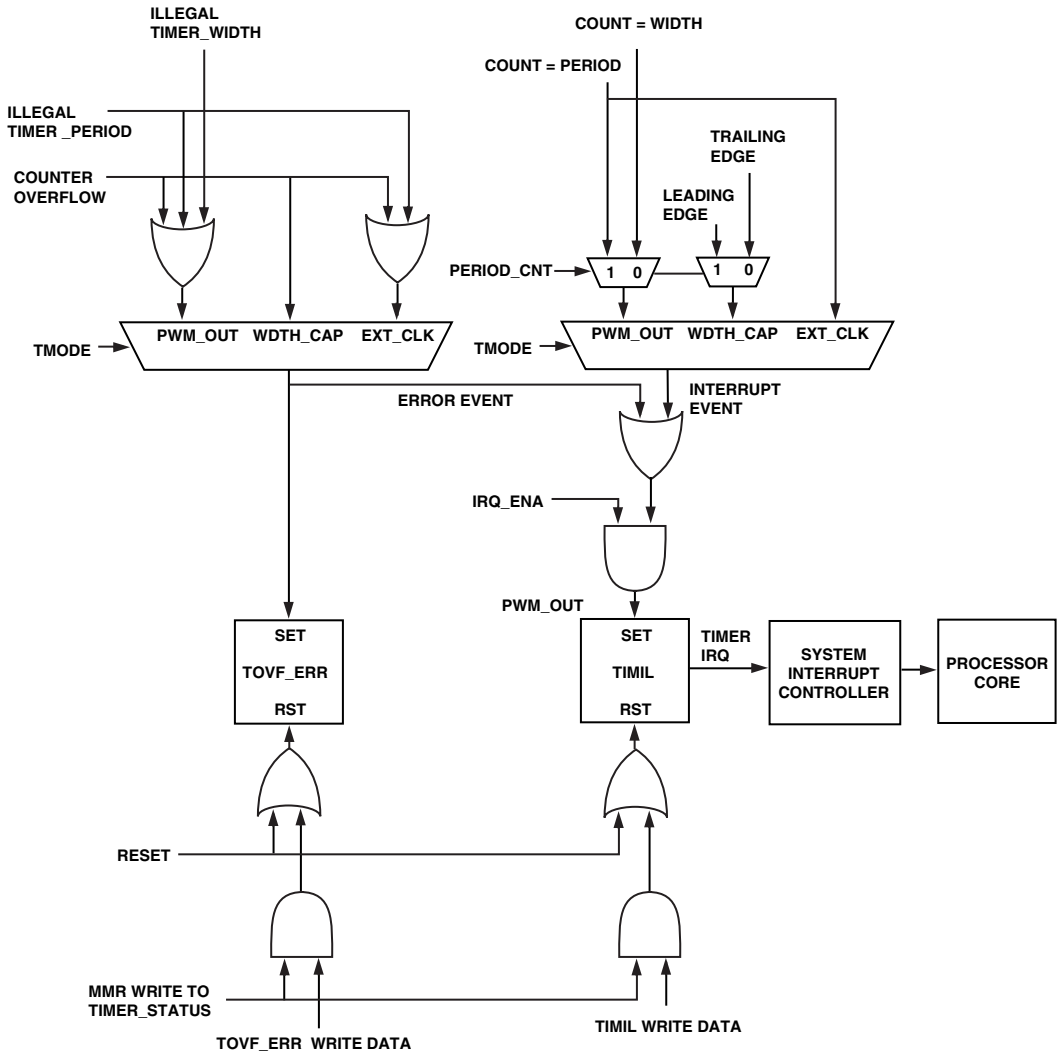


Figure 10-2. Timers Interrupt Structure

To enable interrupt generation, set the IRQ_ENA bit and unmask the interrupt source in the IMASK and SIC_IMASK registers. To poll the TIMIL bit

without interrupt generation, set IRQ_ENA but leave the interrupt masked at the system level. If enabled by IRQ_ENA, interrupt requests are also generated by error conditions as reported by the TOVF_ERR bits.

The system interrupt controller enables flexible interrupt handling. All timers may or may not share the same CEC interrupt channel, so that a single interrupt routine services more than one timer. In PWM mode, multiple timers may run with the same period settings and issue their interrupt requests simultaneously. In this case, the service routine might clear all TIMIL latch bits at once by writing 0x000F 000F to the TIMER_STATUS register.

If interrupts are enabled, make sure that the interrupt service routine (ISR) clears the TIMIL bit in the TIMER_STATUS register before the RTI instruction executes. This ensures that the interrupt is not reissued. Remember that writes to system registers are delayed. If only a few instructions separate the TIMIL clear command from the RTI instruction, an extra SSYNC instruction may be inserted. In EXT_CLK mode, reset the TIMIL bit in the TIMER_STATUS register at the very beginning of the interrupt service routine to avoid missing any timer events.

Illegal States

Every timer features an error detection circuit. It handles overflow situations but also performs pulse width vs. period plausibility checks. Errors are reported by the TOVF_ERR bits in the TIMER_STATUS register and the ERR_TYP bit field in the individual TIMER_CONFIG registers. Table 10-1 provides a summary of error conditions, using these terms:

- **Startup**. The first clock period during which the timer counter is running after the timer is enabled by writing TIMER_ENABLE.
- **Rollover**. The time when the current count matches the value in TIMER_PERIOD and the counter is reloaded with the value "1".

- **Overflow**. The timer counter was incremented instead of doing a rollover when it was holding the maximum possible count value of 0xFFFF FFFF. The counter does not have a large enough range to express the next greater value and so erroneously loads a new value of 0x0000 0000.
- **Unchanged**. No new error.
    - When ERR_TYP is unchanged, it displays the previously reported error code or 00 if there has been no error since this timer was enabled.
    - When TOVF_ERR is unchanged, it reads "0" if there has been no error since this timer was enabled, or if software has performed a W1C to clear any previous error. If a previous error has not been acknowledged by software, TOVF_ERR reads "1".

Software should read TOVF_ERR to check for an error. If TOVF_ERR is set, software can then read ERR_TYP for more information. Once detected, software should write "1" to clear TOVF_ERR to acknowledge the error.

The following table can be read as: “In mode __ at event __, if TIMER_PERIOD is __ and TIMER_WIDTH is __, then ERR_TYP is __ and TOVF_ERR is __.”

Startup error conditions do not prevent the timer from starting. Similarly, overflow and rollover error conditions do not stop the timer. Illegal cases may cause unwanted behavior of the TMR pin.

Table 10-1. Overview of Illegal States

Mode	Event	TIMER_PERIOD	TIMER_WIDTH	ERR_TYP	TOVF_ERR
PWM_OUT, PERIOD_CNT = 1	Startup (No boundary condition tests performed on TIMER_WIDTH)	== 0	Anything	b#10	Set
		== 1	Anything	b#10	Set
		≥ 2	Anything	No change	No change
	Rollover	== 0	Anything	b#10	Set
		== 1	Anything	b#11	Set
		≥ 2	== 0	b#11	Set
		≥ 2	< TIMER_PERIOD	No change	No change
		≥ 2	≥ TIMER_PERIOD	b#11	Set
	Overflow, not possible unless there is also another error, such as TIMER_PERIOD == 0	Anything	Anything	b#01	Set
PWM_OUT, PERIOD_CNT = 0	Startup	Anything	== 0	b#01	Set
		This case is not detected at startup, but results in an overflow error once the counter counts through its entire range.			
		Anything	≥ 1	No change	No change
	Rollover	Rollover is not possible in this mode.			
	Overflow, not possible unless there is also another error, such as TIMER_WIDTH == 0	Anything	Anything	b#01	Set

Table 10-1. Overview of Illegal States (Continued)

Mode	Event	TIMER_PERIOD	TIMER_WIDTH	ERR_TYP	TOVF_ERR
WDTH_CAP	Startup	TIMER_PERIOD and TIMER_WIDTH are read-only in this mode, no error possible.			
	Rollover	TIMER_PERIOD and TIMER_WIDTH are read-only in this mode, no error possible.			
	Overflow	Anything	Anything	b#01	Set
EXT_CLK	Startup	== 0	Anything	b#10	Set
		≥ 1	Anything	No change	No change
	Rollover	== 0	Anything	b#10	Set
		≥ 1	Anything	No change	No change
	Overflow, not possible unless there is also another error, such as TIMER_PERIOD == 0	Anything	Anything	b#01	Set

Modes of Operation

The following sections provide a functional description of the general-purpose timers in various operating modes.

Pulse Width Modulation (PWM_OUT) Mode

Use the PWM_OUT mode for PWM signal or single-pulse generation, for interval timing or for periodic interrupt generation. Figure 10-3 illustrates PWM_OUT mode.

Setting the TMODE field to b#01 in the TIMER_CONFIG register enables PWM_OUT mode. Here, the TMR pin is an output, but it can be disabled by setting the OUT_DIS bit in the TIMER_CONFIG register.

In PWM_OUT mode, the bits PULSE_HI, PERIOD_CNT, IRQ_ENA, OUT_DIS, CLK_SEL, EMU_RUN, and TOGGLE_HI enable orthogonal functionality. They may be set individually or in any combination, although some combinations are not useful (such as TOGGLE_HI = 1 with OUT_DIS = 1 or PERIOD_CNT = 0).

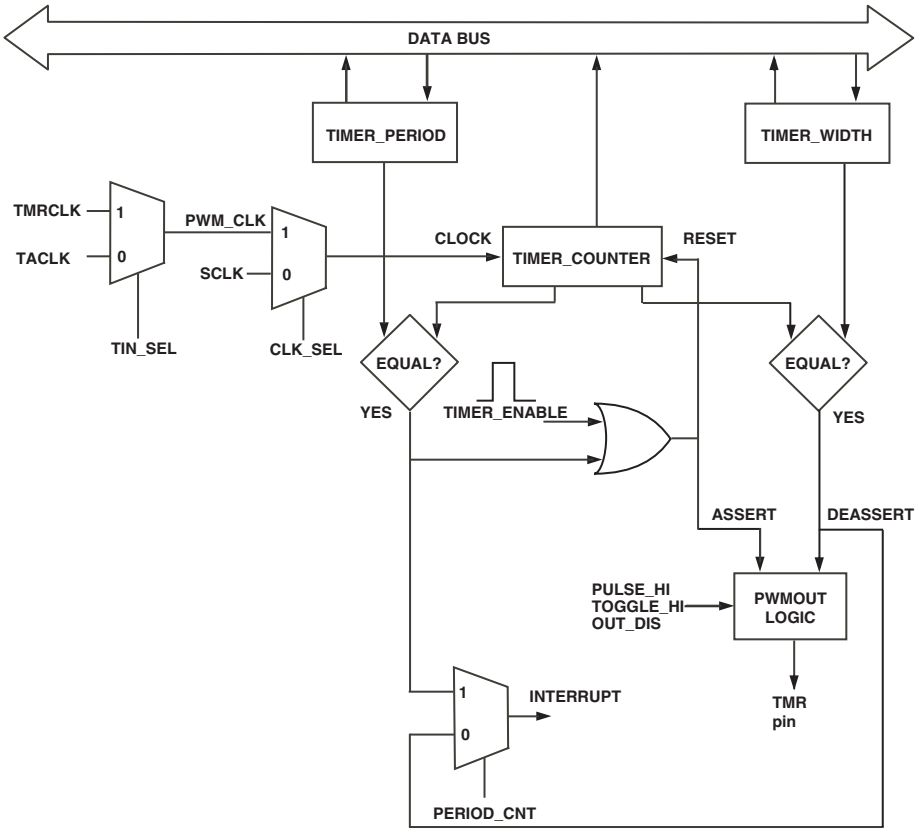


Figure 10-3. Timer Flow Diagram, PWM_OUT Mode

Once a timer has been enabled, the timer counter register is loaded with a starting value. If CLK_SEL = 0, the timer counter starts at 0x1. If CLK_SEL = 1, it is reset to 0x0 as in EXT_CLK mode. The timer counts upward to the value of the timer period register. For either setting of CLK_SEL, when the timer counter equals the timer period, the timer counter is reset to 0x1 on the next clock.

In PWM_OUT mode, the PERIOD_CNT bit controls whether the timer generates one pulse or many pulses. When PERIOD_CNT is cleared (PWM_OUT single pulse mode), the timer uses the TIMER_WIDTH register, generates one asserting and one deasserting edge, then generates an interrupt (if enabled) and stops. When PERIOD_CNT is set (PWM_OUT continuous pulse mode), the timer uses both the TIMER_PERIOD and TIMER_WIDTH registers and generates a repeating (and possibly modulated) waveform. It generates an interrupt (if enabled) at the end of each period and stops only after it is disabled. A setting of PERIOD_CNT = 0 counts to the end of the width; a setting of PERIOD_CNT = 1 counts to the end of the period.

The TIMER_PERIOD and TIMER_WIDTH registers are read-only in some operation modes. Be sure to set the TMODE field in the TIMER_CONFIG register to b#01 before writing to these registers.

Output Pad Disable

The output pin can be disabled in PWM_OUT mode by setting the OUT_DIS bit in the TIMER_CONFIG register. The TMR pin is then three-stated regardless of the setting of PULSE_HI and TOGGLE_HI. This can reduce power consumption when the output signal is not being used. The TMR pin can also be disabled by the function enable and the multiplexer control registers.

Single Pulse Generation

If the PERIOD_CNT bit is cleared, the PWM_OUT mode generates a single pulse on the TMR pin. This mode can also be used to implement a precise delay.

The pulse width is defined by the TIMER_WIDTH register, and the TIMER_PERIOD register is not used. See Figure 10-4.

At the end of the pulse, the timer interrupt latch bit TIMIL is set, and the timer is stopped automatically. No writes to the TIMER_DISABLE register are required in this mode. If the PULSE_HI bit is set, an active high pulse is generated on the TMR pin. If PULSE_HI is not set, the pulse is active low.

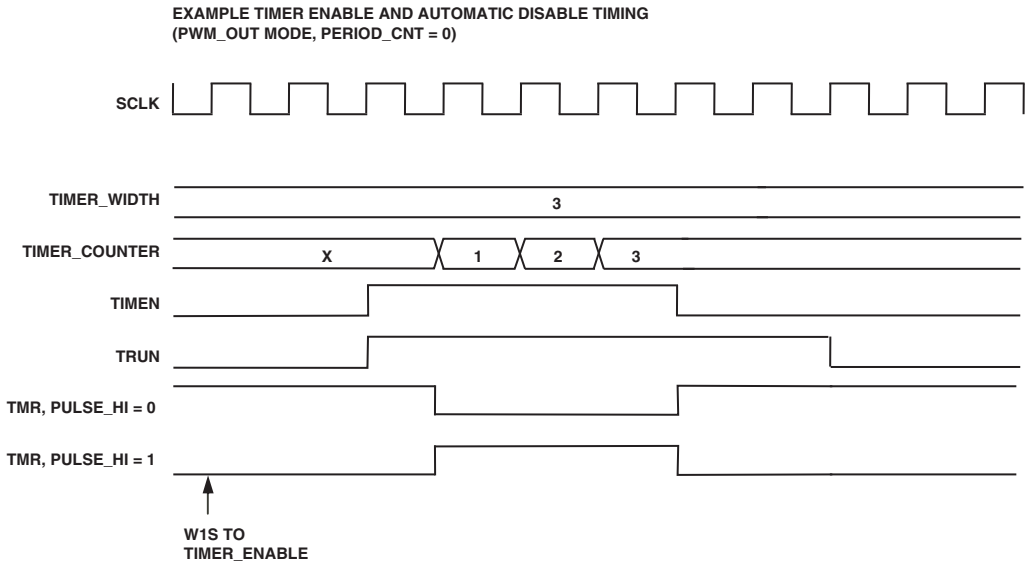


Figure 10-4. Timer Enable and Automatic Disable Timing

The pulse width may be programmed to any value from 1 to (2^{32}-1), inclusive.

Pulse Width Modulation Waveform Generation

If the PERIOD_CNT bit is set, the internally clocked timer generates rectangular signals with well-defined period and duty cycle (PWM patterns). This mode also generates periodic interrupts for real-time signal processing.

The 32-bit TIMER_PERIOD and TIMER_WIDTH registers are programmed with the values required by the PWM signal.

When the timer is enabled in this mode, the TMR pin is pulled to a deasserted state each time the counter equals the value of the pulse width register, and the pin is asserted again when the period expires (or when the timer gets started).

To control the assertion sense of the TMR pin, the PULSE_HI bit in the corresponding TIMER_CONFIG register is used. For a low assertion level, clear this bit. For a high assertion level, set this bit. When the timer is disabled in PWM_OUT mode, the TMR pin is driven to the deasserted level.

Figure 10-5 shows timing details.

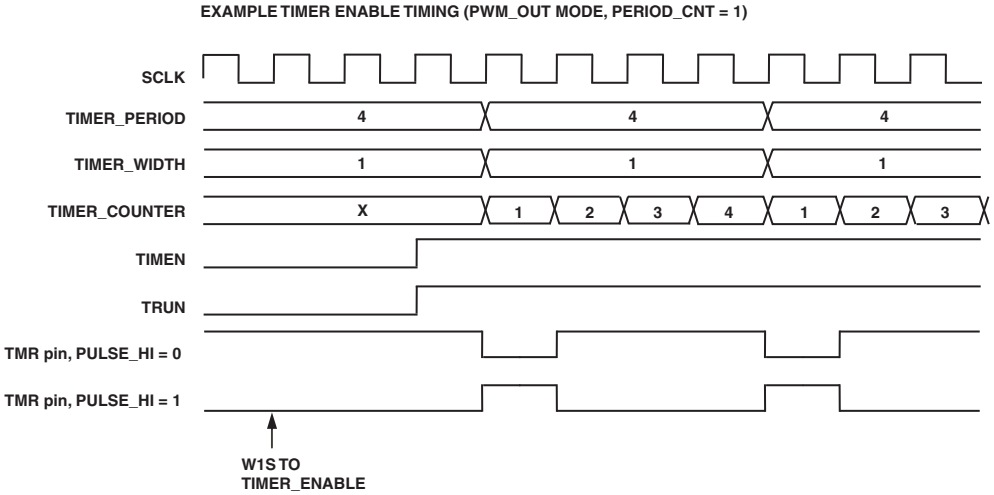


Figure 10-5. Timer Enable Timing

If enabled, a timer interrupt is generated at the end of each period. An interrupt service routine must clear the interrupt latch bit (TIMIL) and might alter period and/or width values. In PWM applications, the software needs to update period and pulse width values while the timer is

running. When software updates either the TIMER_PERIOD or TIMER_WIDTH registers, the new values are held by special buffer registers until the period expires. Then the new period and pulse width values become active simultaneously. Reads from TIMER_PERIOD and TIMER_WIDTH registers return the old values until the period expires.

The TOVF_ERR status bit signifies an error condition in PWM_OUT mode. The TOVF_ERR bit is set if TIMER_PERIOD = 0 or TIMER_PERIOD = 1 at startup, or when the timer counter register rolls over. It is also set if the timer pulse width register is greater than or equal to the timer period register by the time the counter rolls over. The ERR_TYP bits are set when the TOVF_ERR bit is set.

Although the hardware reports an error if the TIMER_WIDTH value equals the TIMER_PERIOD value, this is still a valid operation to implement PWM patterns with 100% duty cycle. If doing so, software must generally ignore the TOVL_ERR flags. Pulse width values greater than the period value are not recommended. Similarly, TIMER_WIDTH = 0 is not a valid operation. Duty cycles of 0% are not supported.

To generate the maximum frequency on the TMR output pin, set the period value to “2” and the pulse width to "1". This makes the pin toggle each SCLK clock, producing a duty cycle of 50%. The period may be programmed to any value from 2 to (2^{32} – 1), inclusive. The pulse width may be programmed to any value from 1 to (period – 1), inclusive.

PULSE_HI Toggle Mode

The waveform produced in PWM_OUT mode with PERIOD_CNT = 1 normally has a fixed assertion time and a programmable deassertion time (via the TIMER_WIDTH register). When two or more timers are running synchro-

nously by the same period settings, the pulses are aligned to the asserting edge as shown in Figure 10-6.

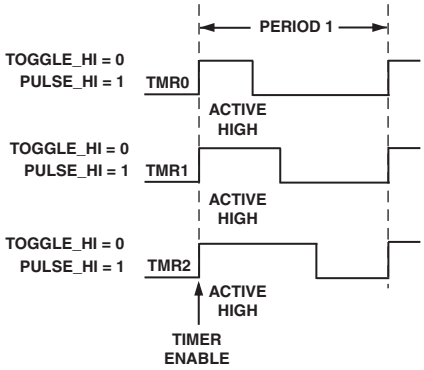


Figure 10-6. Example of Timers With Pulses Aligned to Asserting Edge

The TOGGLE_HI mode enables control of the timing of both the asserting and deasserting edges of the output waveform produced. The phase between the asserting edges of two timer outputs is programmable. The effective state of the PULSE_HI bit alternates every period. The adjacent active low and active high pulses, taken together, create two halves of a symmetrical rectangular waveform. The effective waveform is active high when PULSE_HI is set and active low when PULSE_HI is cleared. The value of the TOGGLE_HI bit has no effect unless the mode is PWM_OUT and PERIOD_CNT = 1.

In TOGGLE_HI mode, when PULSE_HI is set, an active low pulse is generated in the first, third, and all odd-numbered periods, and an active high pulse is generated in the second, fourth, and all even-numbered periods. When PULSE_HI is cleared, an active high pulse is generated in the first, third, and all odd-numbered periods, and an active low pulse is generated in the second, fourth, and all even-numbered periods.

The deasserted state at the end of one period matches the asserted state at the beginning of the next period, so the output waveform only transitions

when Count = Pulse Width. The net result is an output waveform pulse that repeats every two counter periods and is centered around the end of the first period (or the start of the second period).

Figure 10-7 shows an example with three timers running with the same period settings. When software does not alter the PWM settings at run-time, the duty cycle is 50%. The values of the `TIMER_WIDTH` registers control the phase between the signals.

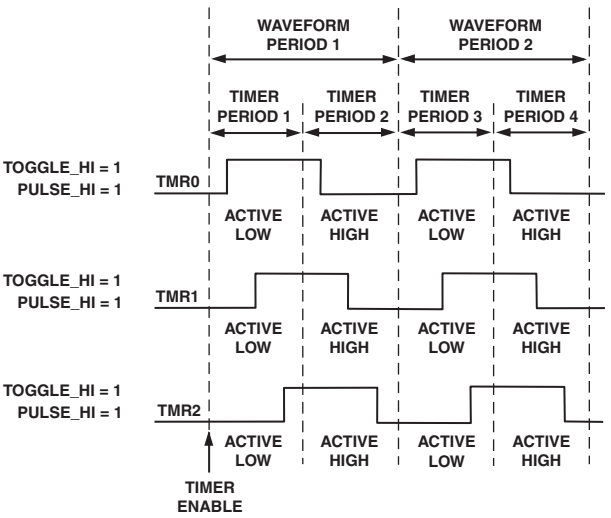


Figure 10-7. Three Timers With Same Period Settings

Similarly, two timers can generate non-overlapping clocks, by center-aligning the pulses while inverting the signal polarity for one of the timers (see Figure 10-8).

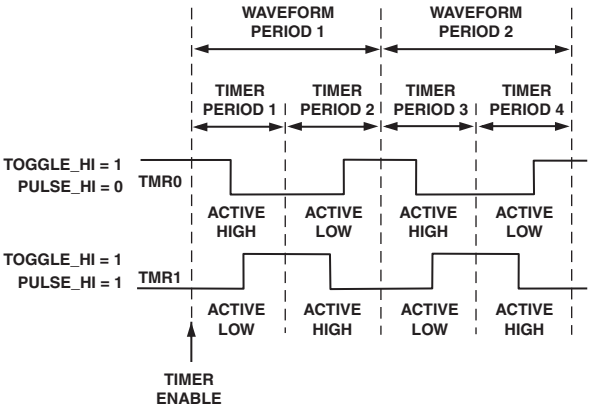


Figure 10-8. Two Timers With Non-overlapping Clocks

When TOGGLE_HI = 0, software updates the TIMER_PERIOD and TIMER_WIDTH registers once per waveform period. When TOGGLE_HI = 1, software updates the TIMER_PERIOD and TIMER_WIDTH registers twice per waveform. Period values are half as large. In odd-numbered periods, write (Period – Width) instead of Width to the TIMER_WIDTH register in order to obtain center-aligned pulses.

For example, if the pseudo-code when TOGGLE_HI = 0 is:

```
int period, width;
for (;;) {
     period = generate_period(...) ;
     width = generate_width(...) ;

     waitfor (interrupt) ;

     write (TIMER_PERIOD, period) ;
```

```
     write (TIMER_WIDTH, width) ;
}
```

Then when `TOGGLE_HI = 1`, the pseudo-code would be:

```
int period, width ;
int per1, per2, wid1, wid2 ;

for (;;) {
     period = generate_period(...) ;
     width = generate_width(...) ;

     per1 = period/2 ;
     wid1 = width/2 ;

     per2 = period/2 ;
     wid2 = width/2 ;

     waitfor (interrupt) ;

     write (TIMER_PERIOD, per1) ;
     write (TIMER_WIDTH, per1 - wid1) ;

     waitfor (interrupt) ;

     write (TIMER_PERIOD, per2) ;
     write (TIMER_WIDTH, wid2) ;

}
```

As shown in this example, the pulses produced do not need to be symmetric (`wid1` does not need to equal `wid2`). The period can be offset to adjust the phase of the pulses produced (`per1` does not need to equal `per2`).

The `TRUN` bit in the `TIMER_STATUS` register is updated only at the end of even-numbered periods in `TOGGLE_HI` mode. When `TIMER_DISABLE` is writ-

ten to "1", the current pair of counter periods (one waveform period) completes before the timer is disabled.

As when TOGGLE_HI = 0, errors are reported if the TIMER_PERIOD register is either set to “0” or "1", or when the width value is greater than or equal to the period value.

Externally Clocked PWM_OUT

By default, the timer is clocked internally by SCLK. Alternatively, if the CLK_SEL bit in the TIMER_CONFIG register is set, the timer is clocked by PWM_CLK. The PWM_CLK is normally input from the TACLK pin, but may be taken from the common TMRCLK pin regardless of whether the timers are configured to work with the PPI. Different timers may receive different signals on their PWM_CLK inputs, depending on configuration. As selected by the PERIOD_CNT bit, the PWM_OUT mode either generates pulse width modulation waveforms or generates a single pulse with pulse width defined by the TIMER_WIDTH register.

When CLK_SEL is set, the counter resets to 0x0 at startup and increments on each rising edge of PWM_CLK. The TMR pin transitions on rising edges of PWM_CLK. There is no way to select the falling edges of PWM_CLK. In this mode, the PULSE_HI bit controls only the polarity of the pulses produced. The timer interrupt may occur slightly before the corresponding edge on the TMR pin (the interrupt occurs on an SCLK edge, the pin transitions on a later PWM_CLK edge). It is still safe to program new period and pulse width values as soon as the interrupt occurs. After a period expires, the counter rolls over to a value of 0x1.

The PWM_CLK clock waveform is not required to have a 50% duty cycle, but the minimum PWM_CLK clock low time is one SCLK period, and the minimum PWM_CLK clock high time is one SCLK period. This implies the maximum PWM_CLK clock frequency is SCLK/2.

The alternate timer clock inputs (TACLK) are enabled when a timer is in PWM_OUT mode with CLK_SEL = 1 and TIN_SEL = 0, without regard to the content of the multiplexer control and function enable registers.

Using PWM_OUT Mode With the PPI

Some timers may be used to generate frame sync signals for certain PPI modes. For detailed instructions on how to configure the timers for use with the PPI, refer to “Frame Synchronization in GP Modes” in the “Parallel Peripheral Interface” chapter of the ADSP-BF52x Blackfin Processor Hardware Reference.

Stopping the Timer in PWM_OUT Mode

In all PWM_OUT mode variants, the timer treats a disable operation (W1C to TIMER_DISABLE) as a “stop is pending” condition. When disabled, it automatically completes the current waveform and then stops cleanly. This prevents truncation of the current pulse and unwanted PWM patterns at the TMR pin. The processor can determine when the timer stops running by polling for the corresponding TRUN bit in the TIMER_STATUS register to read "0" or by waiting for the last interrupt (if enabled). Note the timer cannot be reconfigured (TIMER_CONFIG cannot be written to a new value) until after the timer stops and TRUN reads "0".

In PWM_OUT single pulse mode (PERIOD_CNT = 0), it is not necessary to write TIMER_DISABLE to stop the timer. At the end of the pulse, the timer stops automatically, the corresponding bit in TIMER_ENABLE (and TIMER_DISABLE) is cleared, and the corresponding TRUN bit is cleared. See Figure 10-4 on page 10-13. To generate multiple pulses, write a "1" to TIMER_ENABLE, wait for the timer to stop, then write another "1" to TIMER_ENABLE.

In continuous PWM generation mode (PWM_OUT, PERIOD_CNT = 1) software can stop the timer by writing to the TIMER_DISABLE register. To prevent the ongoing PWM pattern from being stopped in an unpredictable way, the timer does not stop immediately when the corresponding "1" has been

written to the TIMER_DISABLE register. Rather, the write simply clears the enable latch and the timer still completes the ongoing PWM patterns gracefully. It stops cleanly at the end of the first period when the enable latch is cleared. During this final period the TIMEN bit returns "0", but the TRUN bit still reads as a "1".

If the TRUN bit is not cleared explicitly, and the enable latch can be cleared and re-enabled all before the end of the current period will continue to run as if nothing happened. Typically, software should disable a PWM_OUT timer and then wait for it to stop itself.

Figure 10-9 shows detailed timing.

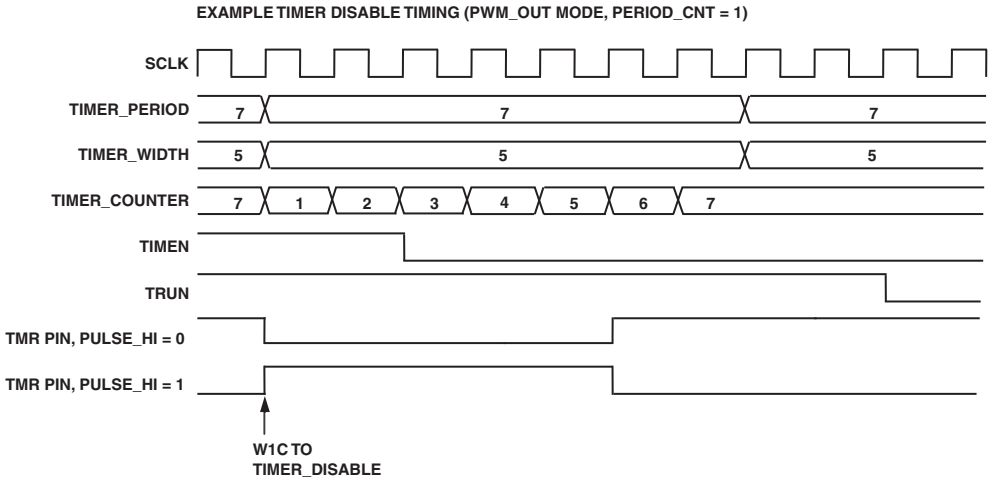


Figure 10-9. Timer Disable Timing

If necessary, the processor can force a timer in PWM_OUT mode to abort immediately. Do this by first writing a "1" to the corresponding bit in TIMER_DISABLE, and then writing a "1" to the corresponding TRUN bit in TIMER_STATUS. This stops the timer whether the pending stop was waiting for the end of the current period (PERIOD_CNT = 1) or the end of the cur-

rent pulse width (PERIOD_CNT = 0). This feature may be used to regain immediate control of a timer during an error recovery sequence.

Use this feature carefully, because it may corrupt the PWM pattern generated at the TMR pin.

When a timer is disabled, the TIMER_COUNTER register retains its state; when a timer is re-enabled, the timer counter is reinitialized based on the operating mode. The TIMER_COUNTER register is read-only. Software cannot overwrite or preset the timer counter value directly.

Pulse Width Count and Capture (WDTH_CAP) Mode

Use the WDTH_CAP mode, often simply called "capture mode," to measure pulse widths on the TMR or TACI input pins, or to "receive" PWM signals. Figure 10-10 shows a flow diagram for WDTH_CAP mode.

In WDTH_CAP mode, the TMR pin is an input pin. The internally clocked timer is used to determine the period and pulse width of externally applied rectangular waveforms. Setting the TMODE field to b#10 in the TIMER_CONFIG register enables this mode.

When enabled in this mode, the timer resets the count in the TIMER_COUNTER register to 0x0000 0001 and does not start counting until it detects a leading edge on the TMR pin.

When the timer detects the first leading edge, it starts incrementing. When it detects a trailing edge of a waveform, the timer captures the current 32-bit value of the TIMER_COUNTER register into the width buffer. At the next leading edge, the timer transfers the current 32-bit value of the TIMER_COUNTER register into the period buffer. The count register is reset to 0x0000 0001 again, and the timer continues counting and capturing until it is disabled.

In this mode, software can measure both the pulse width and the pulse period of a waveform. To control the definition of leading edge and trail-

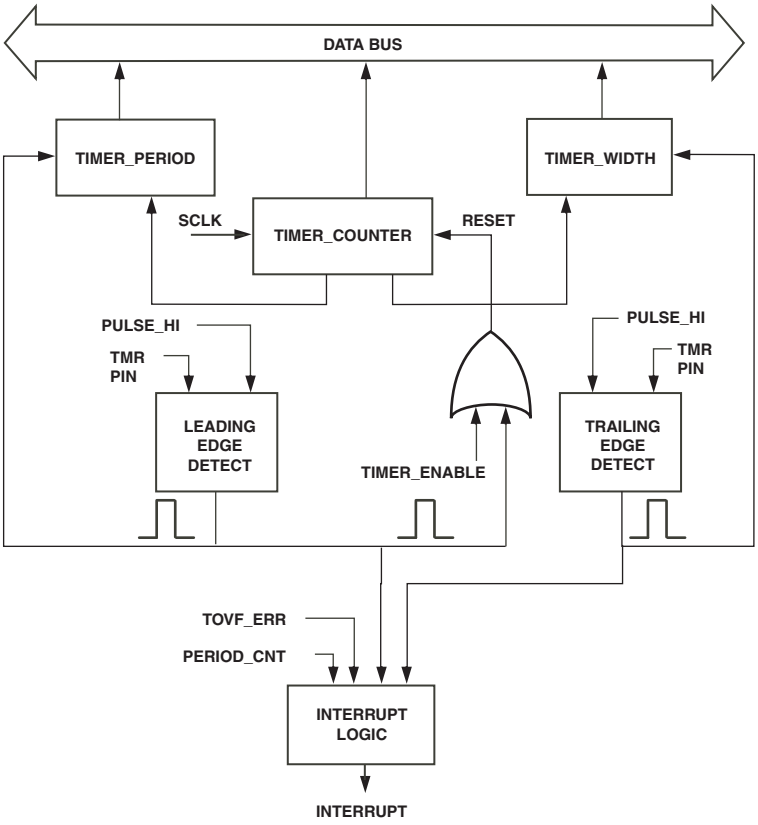


Figure 10-10. Timer Flow Diagram, WDTH_CAP Mode

ing edge of the TMR pin, the PULSE_HI bit in the TIMER_CONFIG register is set or cleared. If the PULSE_HI bit is cleared, the measurement is initiated by a falling edge, the content of the counter register is captured to the pulse width buffer on the rising edge, and to the period buffer on the next falling edge. When the PULSE_HI bit is set, the measurement is initiated by a rising edge, the counter value is captured to the pulse width buffer on the falling edge, and to the period buffer on the next rising edge.

In WDTH_CAP mode, these three events always occur at the same time:

1. The TIMER_PERIOD register is updated from the period buffer.
2. The TIMER_WIDTH register is updated from the width buffer.
3. The TIMIL bit gets set (if enabled) but does not generate an error.

The PERIOD_CNT bit in the TIMER_CONFIG register controls the point in time at which this set of transactions is executed. Taken together, these three events are called a measurement report. The TOVF_ERR bit does not get set at a measurement report. A measurement report occurs, at most, once per input signal period.

The current timer counter value is always copied to the width buffer and period buffer registers at the trailing and leading edges of the input signal, respectively, but these values are not visible to software. A measurement report event samples the captured values into visible registers and sets the timer interrupt to signal that TIMER_PERIOD and TIMER_WIDTH are ready to be read. When the PERIOD_CNT bit is set, the measurement report occurs just after the period buffer captures its value (at a leading edge). When the PERIOD_CNT bit is cleared, the measurement report occurs just after the width buffer captures its value (at a trailing edge).

If the PERIOD_CNT bit is set and a leading edge occurred (see Figure 10-11), then the TIMER_PERIOD and TIMER_WIDTH registers report the pulse period and pulse width measured in the period that just ended. If the PERIOD_CNT bit is cleared and a trailing edge occurred (see Figure 10-12), then the TIMER_WIDTH register reports the pulse width measured in the pulse that just ended, but the TIMER_PERIOD register reports the pulse period measured at the end of the previous period.

If the PERIOD_CNT bit is cleared and the first trailing edge occurred, then the first period value has not yet been measured at the first measurement report, so the period value is not valid. Reading the TIMER_PERIOD value in this case returns "0", as shown in Figure 10-12. To measure the pulse width of a waveform that has only one leading edge and one trailing edge,

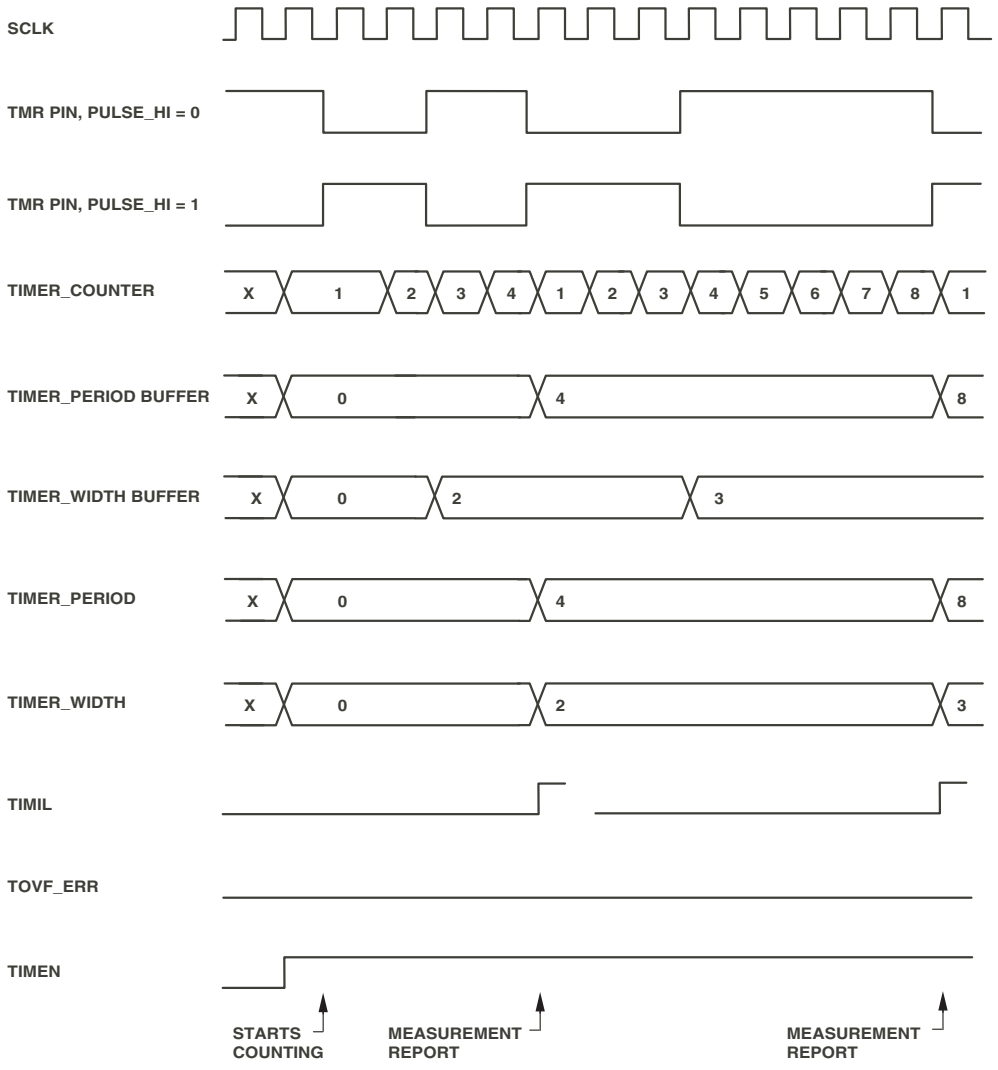
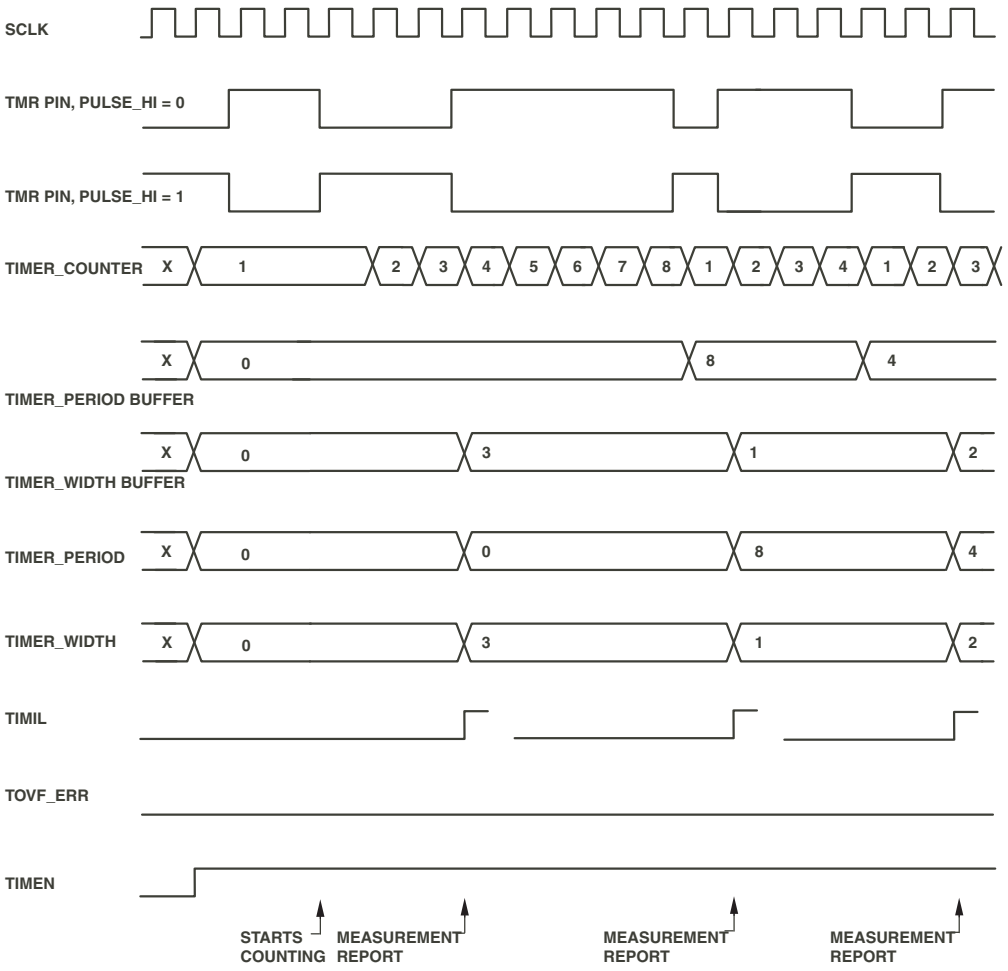


Figure 10-11. Example of Period Capture Measurement Report Timing (WDTH_CAP mode, PERIOD_CNT = 1)



NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMR PIN EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 10-12. Example of Width Capture Measurement Report Timing (WDTH_CAP mode, PERIOD_CNT = 0)

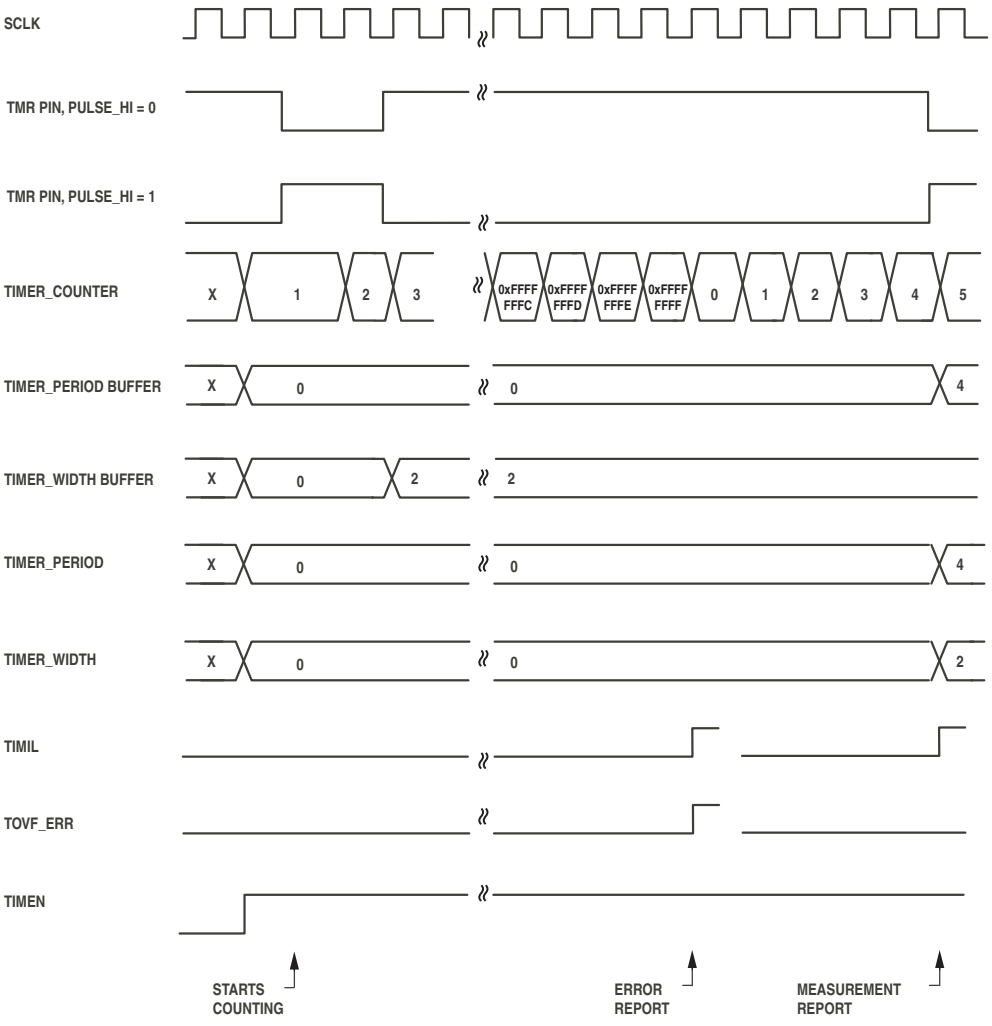
set PERIOD_CNT = 0. If PERIOD_CNT = 1 for this case, no period value is captured in the period buffer. Instead, an error report interrupt is generated (if enabled) when the counter range is exceeded and the counter wraps

around. In this case, both TIMER_WIDTH and TIMER_PERIOD read "0" (because no measurement report occurred to copy the value captured in the width buffer to TIMER_WIDTH). See the first interrupt in Figure 10-13.

When using the PERIOD_CNT = 0 mode described above to measure the width of a single pulse, it is recommended to disable the timer after taking the interrupt that ends the measurement interval. If desired, the timer can then be reenabled as appropriate in preparation for another measurement. This procedure prevents the timer from free-running after the width measurement, and from logging errors generated by the timer count overflowing.

A timer interrupt (if enabled) is generated if the TIMER_COUNTER register wraps around from 0xFFFF FFFF to "0" in the absence of a leading edge. At that point, the TOVF_ERR bit in the TIMER_STATUS register and the ERR_TYP bits in the TIMER_CONFIG register are set, indicating a count overflow due to a period greater than the counter's range. This is called an error report. When a timer generates an interrupt in WDTH_CAP mode, either an error has occurred (an error report) or a new measurement is ready to be read (a measurement report), but never both at the same time. The TIMER_PERIOD and TIMER_WIDTH registers are never updated at the time an error is signaled.

Refer to Figure 10-13 and Figure 10-14 for more information.



NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMR PIN EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 10-13. Example Timing for Period Overflow Followed by Period Capture (`WDTH_CAP` mode, `PERIOD_CNT` = 1)

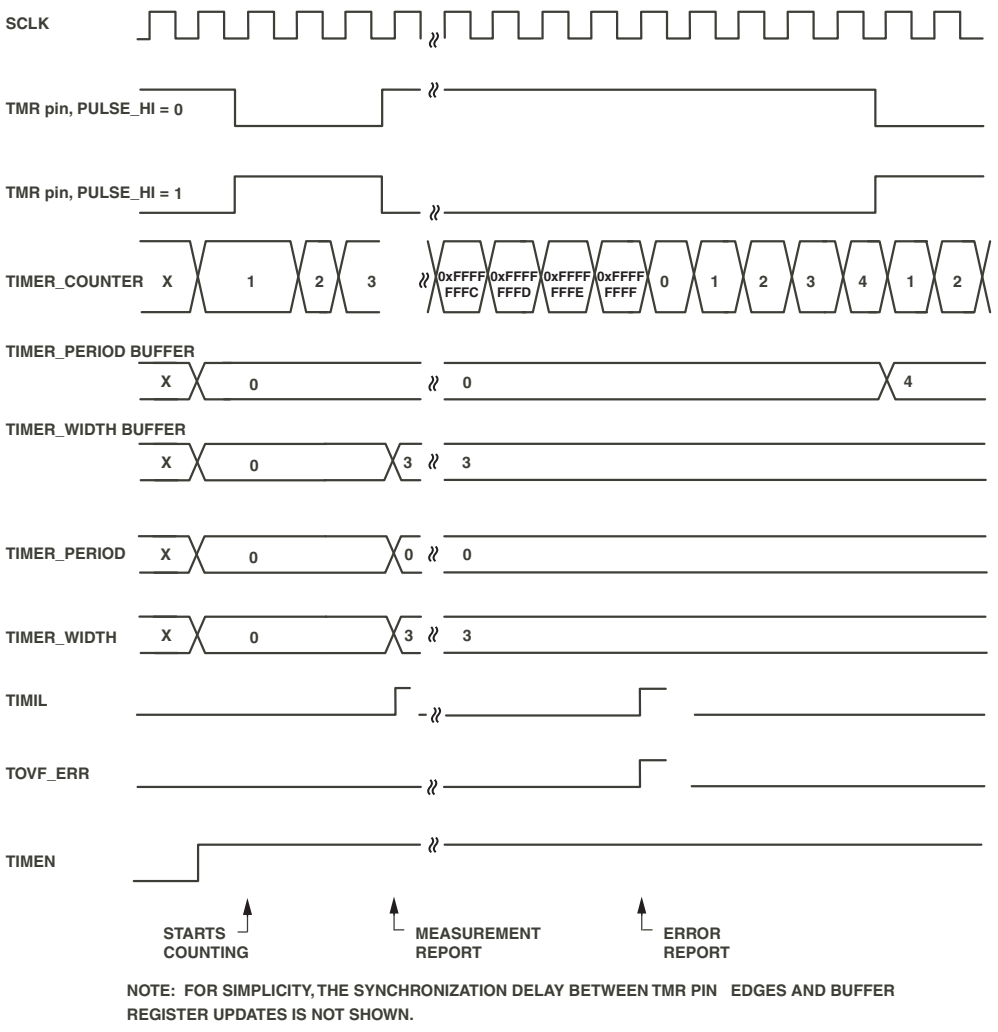


Figure 10-14. Example Timing for Width Capture Followed by Period Overflow (`WDTH_CAP` mode, `PERIOD_CNT` = 0)

Both TIMIL and TOVF_ERR are sticky bits, and software must explicitly clear them. If the timer overflowed and PERIOD_CNT = 1, neither the TIMER_PERIOD nor the TIMER_WIDTH register were updated. If the timer overflowed and PERIOD_CNT = 0, the TIMER_PERIOD and TIMER_WIDTH registers were updated only if a trailing edge was detected at a previous measurement report.

Software can count the number of error report interrupts between measurement report interrupts to measure input signal periods longer than 0xFFFF FFFF. Each error report interrupt adds a full 2^{32} SCLK counts to the total for the period, but the width is ambiguous. For example, in Figure 10-13 the period is 0x1 0000 0004 but the pulse width could be either 0x0 0000 0002 or 0x1 0000 0002.

The waveform applied to the TMR pin is not required to have a 50% duty cycle, but the minimum TMR pin low time is one SCLK period and the minimum TMR pin high time is one SCLK period. This implies the maximum TMR pin input frequency is SCLK/2 with a 50% duty cycle. Under these conditions, the WDTH_CAP mode timer would measure Period = 2 and Pulse Width = 1.

Autobaud Mode

On some devices, in WDTH_CAP mode, some of the timers can provide autobaud detection for the Universal Asynchronous Receiver/Transmitter (UART) interface(s). The TIN_SEL bit in the TIMER_CONFIG register causes the timer to sample the TACI pin instead of the TMR pin when enabled for WDTH_CAP mode. Autobaud detection can be used for initial bit rate negotiations as well as for detection of bit rate drifts while the interface is in operation.

External Event (EXT_CLK) Mode

Use the EXT_CLK mode (sometimes referred to as the counter mode) to count external events—that is, signal edges on the TMR pin (which is an

input in this mode). Figure 10-15 shows a flow diagram for EXT_CLK mode.

The timer works as a counter clocked by an external source, which can also be asynchronous to the system clock. The current count in TIMER_COUNTER represents the number of leading edge events detected. Setting the TMODE field to b#11 in the TIMER_CONFIG register enables this mode. The TIMER_PERIOD register is programmed with the value of the maximum timer external count.

The waveform applied to the TMR pin is not required to have a 50% duty cycle, but the minimum TMR low time is one SCLK period, and the minimum TMR high time is one SCLK period. This implies the maximum TMR pin input frequency is SCLK/2.

Period may be programmed to any value from 1 to (2^{32} – 1), inclusive.

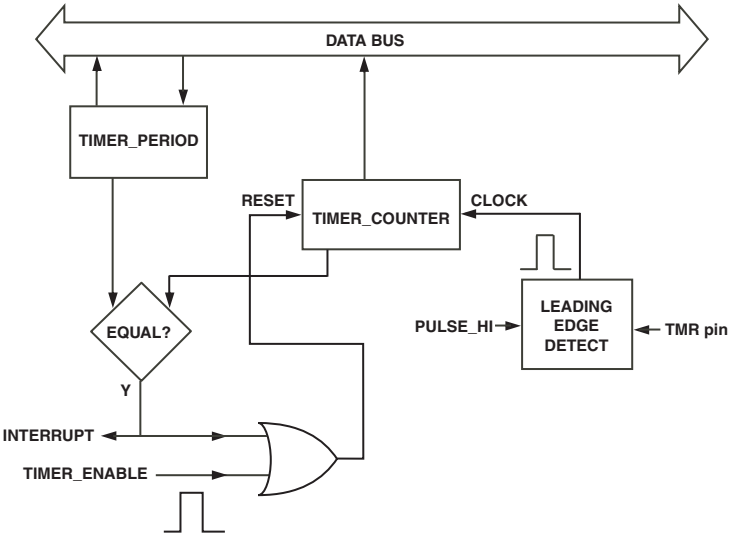


Figure 10-15. Timer Flow Diagram, EXT_CLK Mode

After the timer has been enabled, it resets the TIMER_COUNTER register to 0x0 and then waits for the first leading edge on the TMR pin. This edge causes the TIMER_COUNTER register to be incremented to the value 0x1. Every subsequent leading edge increments the count register. After reaching the period value, the TIMIL bit is set, and an interrupt is generated. The next leading edge reloads the TIMER_COUNTER register again with 0x1. The timer continues counting until it is disabled. The PULSE_HI bit determines whether the leading edge is rising (PULSE_HI set) or falling (PULSE_HI cleared).

The configuration bits TIN_SEL and PERIOD_CNT have no effect in this mode. The TOVF_ERR and ERR_TYP bits are set if the TIMER_COUNTER register wraps around from 0xFFFF FFFF to "0" or if Period = "0" at startup or when the TIMER_COUNTER register rolls over (from Count = Period to Count = 0x1). The TIMER_WIDTH register is unused.

Programming Model

The architecture of the timer block enables any of the timers within this block to work individually or synchronously along with others as a group of timers. Regardless of the operating mode, the programming model is always straightforward. Because of the error checking mechanism, always follow this order when enabling timers:

1. Set timer mode.
2. Write TIMER_WIDTH and TIMER_PERIOD registers as applicable.
3. Enable timer.

If this order is not followed, the plausibility check may fail because of undefined width and period values, or writes to TIMER_WIDTH and TIMER_PERIOD may result in an error condition, because the registers are read-only in some modes. The timer may not start as expected.

If in PWM_OUT mode the PWM patterns of the second period differ from the patterns of the first one, the initialization sequence above might become:

1. Set timer mode to PWM_OUT.
2. Write first TIMER_WIDTH and TIMER_PERIOD value pair.
3. Enable timer.
4. Immediately write second TIMER_WIDTH and TIMER_PERIOD value pair.

Hardware ensures that the buffered width and period values become active when the first period expires.

Once started, timers require minimal interaction with software, which is usually performed by an interrupt service routine. In PWM_OUT mode software must update the pulse width and/or settings as required. In WDTH_CAP mode it must store captured values for further processing. In any case, the service routine should clear the TIMIL bits of the timers it controls.

Timer Registers

The timer peripheral module provides general-purpose timer functionality. It consists of multiple identical timer units.

Each timer provides four registers:

- TIMER_CONFIG[15:0] – timer configuration register
- TIMER_WIDTH[31:0] – timer pulse width register
- TIMER_PERIOD[31:0] – timer period register
- TIMER_COUNTER[31:0] – timer counter register

Additionally, three registers are shared between the timers within a block:

- TIMER_ENABLE[15:0] – timer enable register
- TIMER_DISABLE[15:0] – timer disable register
- TIMER_STATUS[31:0] – timer status register

The size of accesses is enforced. A 32-bit access to a TIMER_CONFIG register or a 16-bit access to a TIMER_WIDTH, TIMER_PERIOD, or TIMER_COUNTER register results in a memory-mapped register (MMR) error. Both 16- and 32-bit accesses are allowed for the TIMER_ENABLE, TIMER_DISABLE, and TIMER_STATUS registers. On a 32-bit read of one of the 16-bit registers, the upper word returns all 0s.

Timer Enable Register (TIMER_ENABLE)

Figure 10-16 shows an example of the TIMER_ENABLE register for a product with eight timers. The register allows simultaneous enabling of multiple timers so that they can run synchronously. For each timer there is a single W1S control bit. Writing a "1" enables the corresponding timer; writing a "0" has no effect. The bits can be set individually or in any combination. A read of the TIMER_ENABLE register shows the status of the enable for the corresponding timer. A "1" indicates that the timer is enabled. All unused bits return "0" when read.

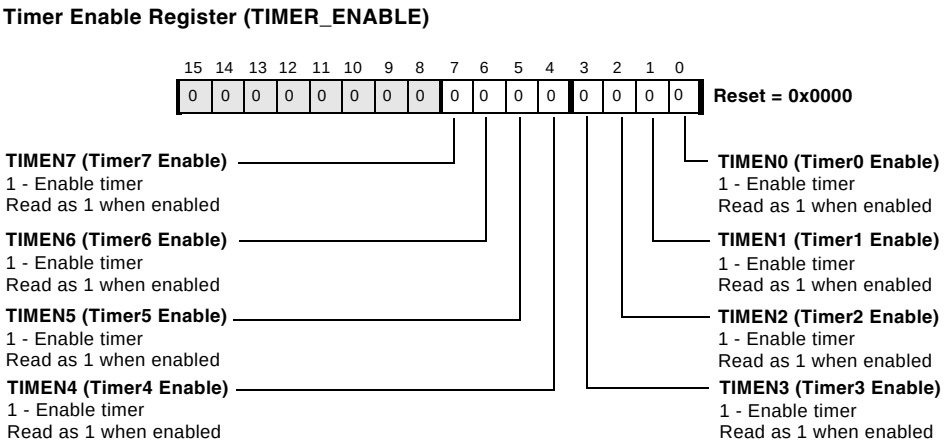
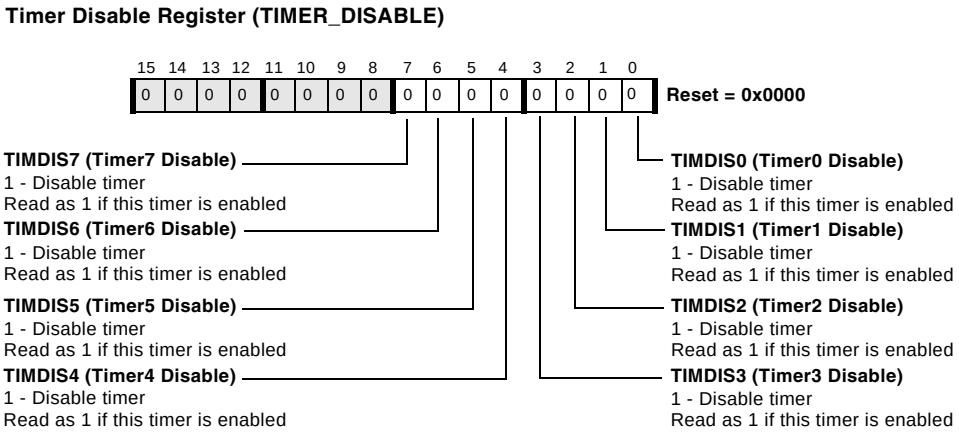

Timer Enable Register (TIMER_ENABLE)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
TIMEN7 (Timer7 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN6 (Timer6 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN5 (Timer5 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN4 (Timer4 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN0 (Timer0 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN1 (Timer1 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN2 (Timer2 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN3 (Timer3 Enable)
1 - Enable timer
Read as 1 when enabled
This diagram shows an example configuration for eight timers. Different products have different numbers of timers.

Figure 10-16. Timer Enable Register

Timer Disable Register (TIMER_DISABLE)

Figure 10-17 shows an example of the TIMER_DISABLE register for a product with eight timers. The register allows simultaneous disabling of multiple timers. For each timer there is a single W1C control bit. Writing a "1" disables the corresponding timer; writing a "0" has no effect. The bits can be cleared individually or in any combination. A read of the TIMER_DISABLE register returns a value identical to a read of the TIMER_ENABLE register. A "1" indicates that the timer is enabled. All unused bits return "0" when read.


Timer Disable Register (TIMER_DISABLE)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
TIMDIS7 (Timer7 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS6 (Timer6 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS5 (Timer5 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS4 (Timer4 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS0 (Timer0 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS1 (Timer1 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS2 (Timer2 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS3 (Timer3 Disable)
1 - Disable timer
Read as 1 if this timer is enabled

This diagram shows an example configuration for eight timers. Different products have different numbers of timers.

Figure 10-17. Timer Disable Register

In PWM_OUT mode, a write of a "1" to TIMER_DISABLE does not stop the corresponding timer immediately. Rather, the timer continues running and stops cleanly at the end of the current period (if PERIOD_CNT = 1) or pulse (if PERIOD_CNT = 0). If necessary, the processor can force a timer in PWM_OUT mode to stop immediately by first writing a "1" to the corresponding bit in TIMER_DISABLE, and then writing a "1" to the corresponding TRUN bit in TIMER_STATUS. See "Stopping the Timer in PWM_OUT Mode" on page 10-21.

In WDTH_CAP and EXT_CLK modes, a write of a "1" to TIMER_DISABLE stops the corresponding timer immediately.

Timer Status Register (TIMER_STATUS)

The TIMER_STATUS register indicates the status of the timers and is used to check the status of multiple timers with a single read. Status bits are sticky and W1C. The TRUN bits can clear themselves, which they do when a PWM_OUT mode timer stops at the end of a period. During a TIMER_STATUS

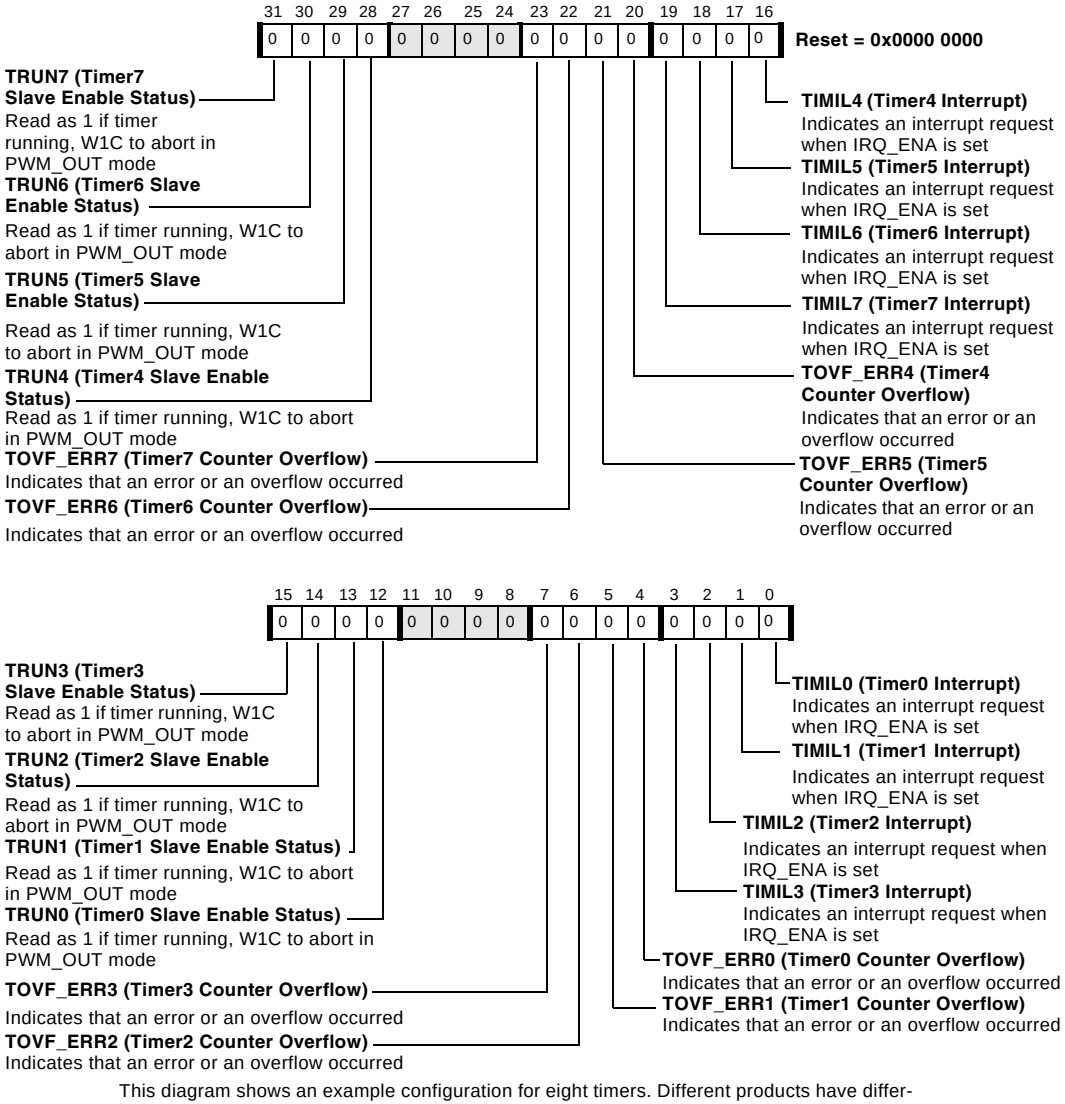
register read access, all reserved or unused bits return a "0". Figure 10-18 on page 10-39 shows an example of the TIMER_STATUS register for a product with eight timers.

For detailed behavior and usage of the TRUN bit see "Stopping the Timer in PWM_OUT Mode" on page 10-21. Writing the TRUN bits has no effect in other modes or when a timer has not been enabled. Writing the TRUN bits to "1" in PWM_OUT mode has no effect on a timer that has not first been disabled.

Error conditions are explained in "Illegal States" on page 10-7.

Timer Status Register (TIMER_STATUS)

All bits are W1C


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
TRUN7 (Timer7 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TRUN6 (Timer6 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TRUN5 (Timer5 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TRUN4 (Timer4 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TOVF_ERR7 (Timer7 Counter Overflow)
Indicates that an error or an overflow occurred
TOVF_ERR6 (Timer6 Counter Overflow)
Indicates that an error or an overflow occurred
TIMIL4 (Timer4 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TIMIL5 (Timer5 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TIMIL6 (Timer6 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TIMIL7 (Timer7 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TOVF_ERR4 (Timer4 Counter Overflow)
Indicates that an error or an overflow occurred
TOVF_ERR5 (Timer5 Counter Overflow)
Indicates that an error or an overflow occurred
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
TRUN3 (Timer3 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TRUN2 (Timer2 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TRUN1 (Timer1 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TRUN0 (Timer0 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TOVF_ERR3 (Timer3 Counter Overflow)
Indicates that an error or an overflow occurred
TOVF_ERR2 (Timer2 Counter Overflow)
Indicates that an error or an overflow occurred
TIMIL0 (Timer0 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TIMIL1 (Timer1 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TIMIL2 (Timer2 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TIMIL3 (Timer3 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TOVF_ERR0 (Timer0 Counter Overflow)
Indicates that an error or an overflow occurred
TOVF_ERR1 (Timer1 Counter Overflow)
Indicates that an error or an overflow occurred

This diagram shows an example configuration for eight timers. Different products have different numbers of timers, therefore some of the bits may not be applicable to your device.

Figure 10-18. Timer Status Register

Timer Configuration Register (TIMER_CONFIG)

The operating mode for each timer is specified by its TIMER_CONFIG register. The TIMER_CONFIG register, shown in Figure 10-19, may be written only when the timer is not running. After disabling the timer in PWM_OUT mode, make sure the timer has stopped running by checking its TRUN bit in TIMER_STATUS before attempting to reprogram TIMER_CONFIG. The TIMER_CONFIG registers may be read at any time. The ERR_TYP field is read-only. It is cleared at reset and when the timer is enabled.

Each time TOVF_ERR is set, ERR_TYP[1:0] is loaded with a code that identifies the type of error that was detected. This value is held until the next error or timer enable occurs. For an overview of error conditions, see Table 10-1 on page 10-9. The TIMER_CONFIG register also controls the behavior of the TMR pin, which becomes an output in PWM_OUT mode (TMODE = 01) when the OUT_DIS bit is cleared.

When operating the PPI in GP output modes with internal frame syncs, the CLK_SEL and the TIN_SEL bits for the timers involved must be set to "1".

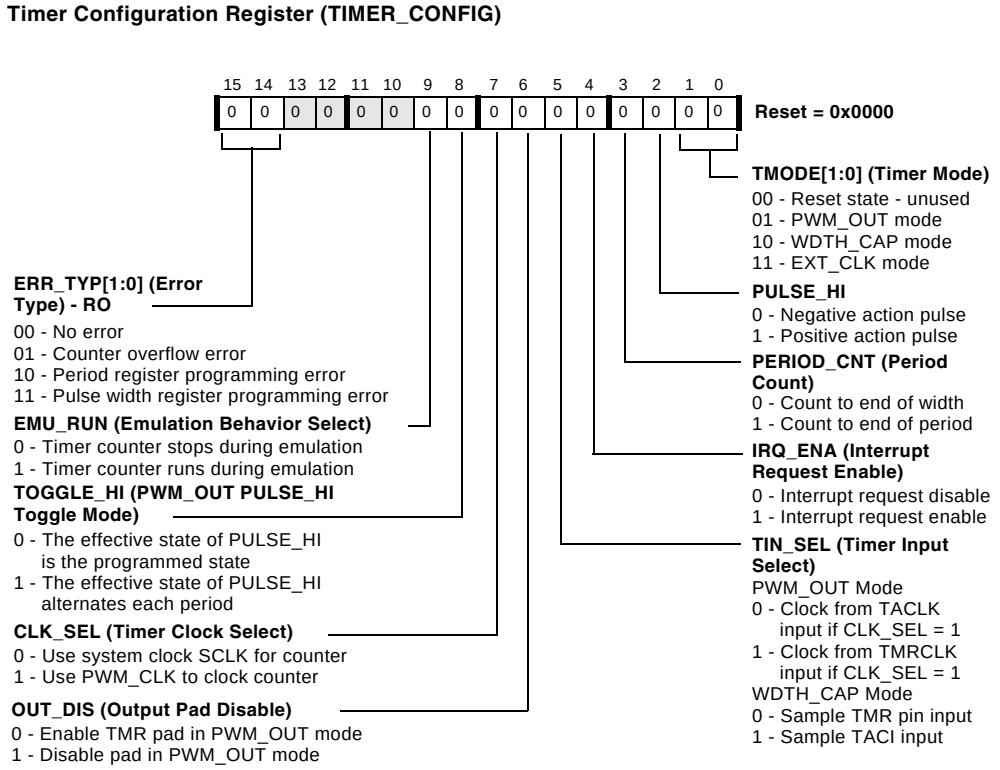

Timer Configuration Register (TIMER_CONFIG)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
TMODE[1:0] (Timer Mode)
00 - Reset state - unused
01 - PWM_OUT mode
10 - WDTH_CAP mode
11 - EXT_CLK mode
PULSE_HI
0 - Negative action pulse
1 - Positive action pulse
PERIOD_CNT (Period Count)
0 - Count to end of width
1 - Count to end of period
IRQ_ENA (Interrupt Request Enable)
0 - Interrupt request disable
1 - Interrupt request enable
TIN_SEL (Timer Input Select)
PWM_OUT Mode
0 - Clock from TACLK input if CLK_SEL = 1
1 - Clock from TMRCLK input if CLK_SEL = 1
WDTH_CAP Mode
0 - Sample TMR pin input
1 - Sample TACI input
ERR_TYP[1:0] (Error Type) - RO
00 - No error
01 - Counter overflow error
10 - Period register programming error
11 - Pulse width register programming error
EMU_RUN (Emulation Behavior Select)
0 - Timer counter stops during emulation
1 - Timer counter runs during emulation
TOGGLE_HI (PWM_OUT PULSE_HI Toggle Mode)
0 - The effective state of PULSE_HI is the programmed state
1 - The effective state of PULSE_HI alternates each period
CLK_SEL (Timer Clock Select)
0 - Use system clock SCLK for counter
1 - Use PWM_CLK to clock counter
OUT_DIS (Output Pad Disable)
0 - Enable TMR pad in PWM_OUT mode
1 - Disable pad in PWM_OUT mode

Figure 10-19. Timer Configuration Register

Timer Counter Register (TIMER_COUNTER)

This read-only register retains its state when disabled. When enabled, the TIMER_COUNTER register is reinitialized by hardware based on configuration and mode. The TIMER_COUNTER register, shown in Figure 10-20, may be read at any time (whether the timer is running or stopped), and it returns an atomic 32-bit value. Depending on the operating mode, the incrementing counter can be clocked by four different sources: SCLK, the TMR pin, the alternative timer clock pin TACLK, or the common TMRCLK pin, which is most likely used as the PPI clock (PPI_CLK).

While the processor core is being accessed by an external emulator debugger, all code execution stops. By default, the TIMER_COUNTER register also halts its counting during an emulation access in order to remain synchronized with the software. While stopped, the count does not advance—in PWM_OUT mode, the TMR pin waveform is “stretched”; in WDTH_CAP mode, measured values are incorrect; in EXT_CLK mode, input events on the TMR pin may be missed. All other timer functions such as register reads and writes, interrupts previously asserted (unless cleared), and the loading of TIMER_PERIOD and TIMER_WIDTH in WDTH_CAP mode remain active during an emulation stop.

Some applications may require the timer to continue counting asynchronously to the emulation-halted processor core. Set the EMU_RUN bit in TIMER_CONFIG to enable this behavior.

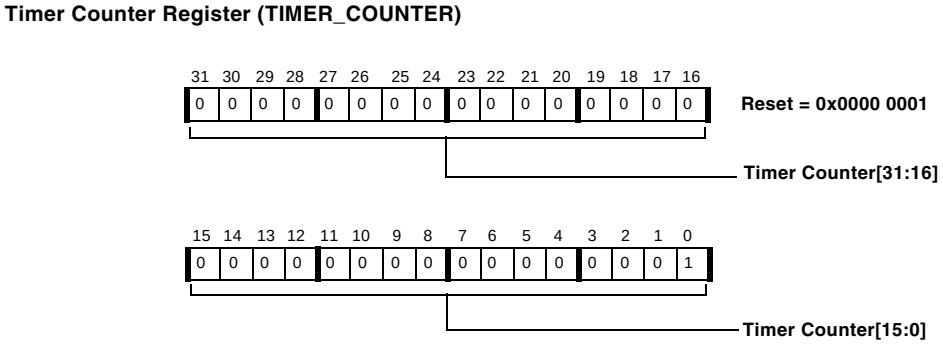


Figure 10-20. Timer Counter Register

Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers

When a timer is enabled and running, and the software writes new values to the TIMER_PERIOD register and the TIMER_WIDTH register, the writes are buffered and do not update the registers until the end of the current period (when TIMER_COUNTER equals TIMER_WIDTH).

Usage of the TIMER_PERIOD register, shown in Figure 10-21, and the TIMER_WIDTH register, shown in Figure 10-22, varies depending on the mode of the timer:

- In PWM_OUT mode, both the TIMER_PERIOD and TIMER_WIDTH register values can be updated “on-the-fly” since the values change simultaneously.
- In WDTH_CAP mode, the timer period and timer pulse width buffer values are captured at the appropriate time. The TIMER_PERIOD and TIMER_WIDTH registers are then updated simultaneously from their respective buffers. Both registers are read-only in this mode.
- In EXT_CLK mode, the TIMER_PERIOD register is writable and can be updated “on-the-fly.” The TIMER_WIDTH register is not used.

If new values are not written to the TIMER_PERIOD register or the TIMER_WIDTH register, the value from the previous period is reused. Writes to the 32-bit TIMER_PERIOD register and TIMER_WIDTH register are atomic; it is not possible for the high word to be written without the low word also being written.

Values written to the TIMER_PERIOD registers or TIMER_WIDTH registers are always stored in the buffer registers. Reads from the TIMER_PERIOD or TIMER_WIDTH registers always return the current, active value of period or pulse width. Written values are not read back until they become active. When the timer is enabled, they do not become active until after the TIMER_PERIOD and TIMER_WIDTH registers are updated from their respective buffers at the end of the current period. See Figure 10-1 on page 10-3.

When the timer is disabled, writes to the buffer registers are immediately copied through to the TIMER_PERIOD or TIMER_WIDTH register so that they will be ready for use in the first timer period. For example, to change the values for the TIMER_PERIOD and/or TIMER_WIDTH registers in order to use a different setting for each of the first three timer periods after the timer is enabled, the procedure to follow is:

1. Program the first set of register values.
2. Enable the timer.
3. Immediately program the second set of register values.
4. Wait for the first timer interrupt.
5. Program the third set of register values.

Each new setting is then programmed when a timer interrupt is received.

In PWM_OUT mode with very small periods (less than 10 counts), there may not be enough time between updates from the buffer registers to write both the TIMER_PERIOD register and the TIMER_WIDTH register. The next period may use one old value and one new value. In order to prevent “pulse width ≥ period” errors,

write the TIMER_WIDTH register before the TIMER_PERIOD register when decreasing the values, and write the TIMER_PERIOD register before the TIMER_WIDTH register when increasing the value.

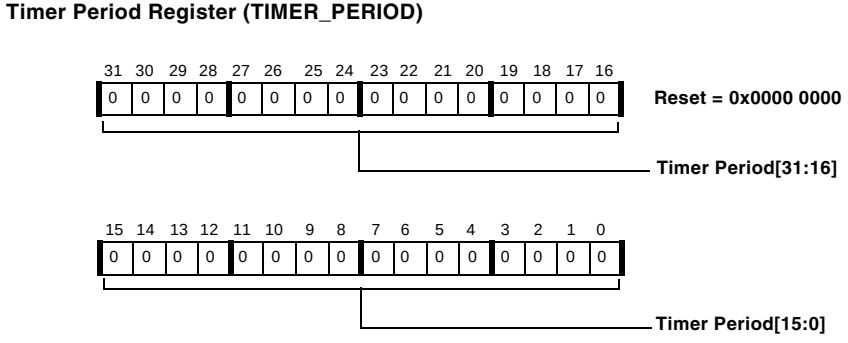


Figure 10-21. Timer Period Register

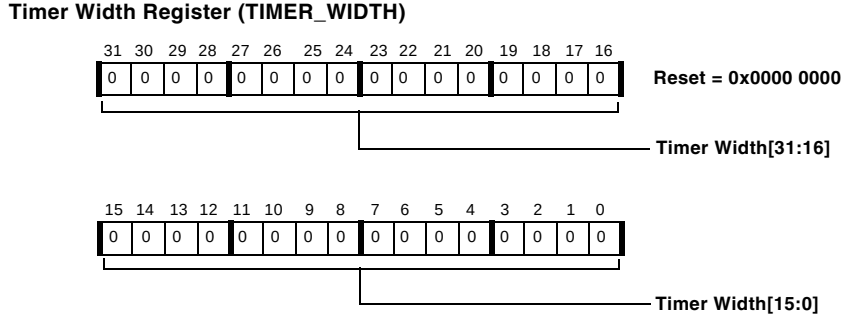


Figure 10-22. Timer Width Register

Summary

Table 10-2 summarizes control bit and register usage in each timer mode.

Table 10-2. Control Bit and Register Usage Chart

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TIMER_ENABLE	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect
TIMER_DISABLE	1 - Disable timer at end of period 0 - No effect	1 - Disable timer 0 - No effect	1 - Disable timer 0 - No effect
TMODE	b#01	b#10	b#11
PULSE_HI	1 - Generate high width 0 - Generate low width	1 - Measure high width 0 - Measure low width	1 - Count rising edges 0 - Count falling edges
PERIOD_CNT	1 - Generate PWM 0 - Single width pulse	1 - Interrupt after measuring period 0 - Interrupt after measuring width	Unused
IRQ_ENA	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt
TIN_SEL	Depends on CLK_SEL: If CLK_SEL = 1, 1 - Count TMRCLK clocks 0 - Count TACLK clocks If CLK_SEL = 0, Unused	1 - Select TACI input 0 - Select TMR pin input	Unused
OUT_DIS	1 - Disable TMR pin 0 - Enable TMR pin	Unused	Unused
CLK_SEL	1 - PWM_CLK clocks timer 0 - SCLK clocks timer	Unused	Unused

Table 10-2. Control Bit and Register Usage Chart (Continued)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TOGGLE_HI	1 - One waveform period every two counter periods 0 - One waveform period every one counter period	Unused	Unused
ERR_TYP	Reports b#00, b#01, b#10, or b#11, as appropriate	Reports b#00 or b#01, as appropriate	Reports b#00, b#01, or b#10, as appropriate
EMU_RUN	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation
TMR Pin	Depends on OUT_DIS: 1 - Three-state 0 - Output	Depends on TIN_SEL: 1 - Unused 0 - Input	Input
Period	R/W: Period value	RO: Period value	R/W: Period value
Width	R/W: Width value	RO: Width value	Unused
Counter	RO: Counts up on SCLK or PWM_CLK	RO: Counts up on SCLK	RO: Counts up on TMR pin event
TRUN	Read: Timer slave enable status Write: 1 - Stop timer if disabled 0 - No effect	Read: Timer slave enable status Write: 1 - No effect 0 - No effect	Read: Timer slave enable status Write: 1 - No effect 0 - No effect

Table 10-2. Control Bit and Register Usage Chart (Continued)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TOVF_ERR	Set at startup or roll-over if period = 0 or 1 Set at rollover if width >= Period Set if counter wraps	Set if counter wraps	Set if counter wraps or set at startup or roll-over if period = 0
IRQ	Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter equals period and PERIOD_CNT = 1 or when counter equals width and PERIOD_CNT = 0 0 - Not set	Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter captures period and PERIOD_CNT = 1 or when counter captures width and PERIOD_CNT = 0 0 - Not set	Depends on IRQ_ENA: 1 - Set when counter equals period or TOVF_ERR set 0 - Not set

Programming Examples

Listing 10-1 configures the port control registers in a way that enables `TMR` pins associated with Port G. This example assumes `TMR1-7` are connected to Port G bits 5–11.

Listing 10-1. Port Setup

```
timer_port_setup:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(PORTG_FER);
    p5.l = lo(PORTG_FER);
    r7.l = PG5|PG6|PG7|PG8|PG9|PG10|PG11;
    w[p5] = r7;
    p5.l = lo(PORTG_MUX);
    r7.l = PFTE;
```

```
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
timer_port_setup.end:
```

Listing 10-2 generates signals on the TMR4 and TMR5 outputs. By default, timer 5 generates a continuous PWM signal with a duty cycle of 50% (period = 0x40 SCLKs, width = 0x20 SCLKs) while the PWM signal generated by timer 4 has the same period but 25% duty cycle (width = 0x10 SCLKs).

If the preprocessor constant SINGLE_PULSE is defined, every TMR pin outputs only a single high pulse of 0x20 (timer 4) and 0x10 SCLKs (timer 5) duration.

In any case the timers are started synchronously and the rising edges are aligned. That is, the pulses are left aligned.

Listing 10-2. Signal Generation

```
// #define SINGLE_PULSE
timer45_signal_generation:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
#ifdef SINGLE_PULSE
    r7.l = PULSE_HI | PWM_OUT;
#else
    r7.l = PERIOD_CNT | PULSE_HI | PWM_OUT;
#endif
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    w[p5 + TIMER4_CONFIG - TIMER_ENABLE] = r7;
    r7 = 0x10 (z);
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r7;
    r7 = 0x20 (z);
```

```
    [p5 + TIMER4_WIDTH - TIMER_ENABLE] = r7;
#ifndef SINGLE_PULSE
    r7 = 0x40 (z);
    [p5 + TIMER5_PERIOD - TIMER_ENABLE] = r7;
    [p5 + TIMER4_PERIOD - TIMER_ENABLE] = r7;
#endif
    r7.l = TIMEN5 | TIMEN4;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
timer45_signal_generation.end:
```

All subsequent examples use interrupts. Thus, Listing 10-3 illustrates how interrupts are generated and how interrupt service routines can be registered. In this example, the timer 5 interrupt is assigned to the IVG12 interrupt channel of the CEC controller.

Listing 10-3. Interrupt Setup

```
timer5_interrupt_setup:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(IMASK);
    p5.l = lo(IMASK);
/* register interrupt service routine */
    r7.h = hi(isr_timer5);
    r7.l = lo(isr_timer5);
    [p5 + EVT12 - IMASK] = r7;
/* unmask IVG12 in CEC */
    r7 = [p5];
    bitset(r7, bitpos(EVT_IVG12));
    [p5] = r7;
/* assign timer 5 IRQ (= IRQ37 in this example) to IVG12 */
    p5.h = hi(SIC_IAR4);
    p5.l = lo(SIC_IAR4);
/*SIC_IAR register mapping is processor dependent*/
```

```
    r7.h = 0xFF5F;
    r7.l = 0xFFFF;
    [p5] = r7;
/* enable timer 5 IRQ */
    p5.h = hi(SIC_IMASK1);
    p5.l = lo(SIC_IMASK1);
/*SIC_IMASK register mapping is processor dependent*/
    r7 = [p5];
    bitset(r7, 5);
    [p5] = r7;
/* enable interrupt nesting */
    (r7:7, p5:5) = [sp++];
    [--sp] = reti;
    rts;
timer5_interrupt_setup.end:
```

The example shown in Listing 10-4 does not drive the TMR pin. It generates periodic interrupt requests every 0x1000 SCLK cycles. If the preprocessor constant SINGLE_PULSE was defined, timer 5 requests an interrupt only once. Unlike in a real application, the purpose of the interrupt service routine shown in this example is just the clearing of the interrupt request and counting interrupt occurrences.

Listing 10-4. Periodic Interrupt Requests

```
// #define SINGLE_PULSE
timer5_interrupt_generation:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
#ifdef SINGLE_PULSE
    r7.l = EMU_RUN | IRQ_ENA | OUT_DIS | PWM_OUT;
#else
    r7.l = EMU_RUN | IRQ_ENA | PERIOD_CNT | OUT_DIS | PWM_OUT;
#endif
```

```
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    r7 = 0x1000 (z);
#ifndef SINGLE_PULSE
    [p5 + TIMER5_PERIOD - TIMER_ENABLE] = r7;
    r7 = 0x1 (z);
#endif
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r7;
    r7.l = TIMEN5;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    r0 = 0 (z);
    rts;
timer5_interrupt_generation.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_STATUS);
    p5.l = lo(TIMER_STATUS);
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5] = r7;
    r0+= 1;
    ssync;
    (r7:7, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```

Listing 10-5 illustrates how two timers can generate two non-overlapping clock pulses as typically required for break-before-make scenarios. Both timers are running in `PWM_OUT` mode with `PERIOD_CNT` = 1 and `PULSE_HI` = 1.

Figure 10-23 explains how the signal waveform represented by the period P and the pulse width W translates to timer period and width values. Table 10-3 summarizes the register writes.

Table 10-3. Register Writes for Non-Overlapping Clock Pulses

Register	Before Enable	After Enable	At IRQ1	At IRQ2
TIMER5_PERIOD	P/2			
TIMER5_WIDTH	P/2 - W/2	W/2	P/2 - W/2	W/2
TIMER4_PERIOD	P	P/2		
TIMER4_WIDTH	P - W/2		W/2	P/2 - W-2

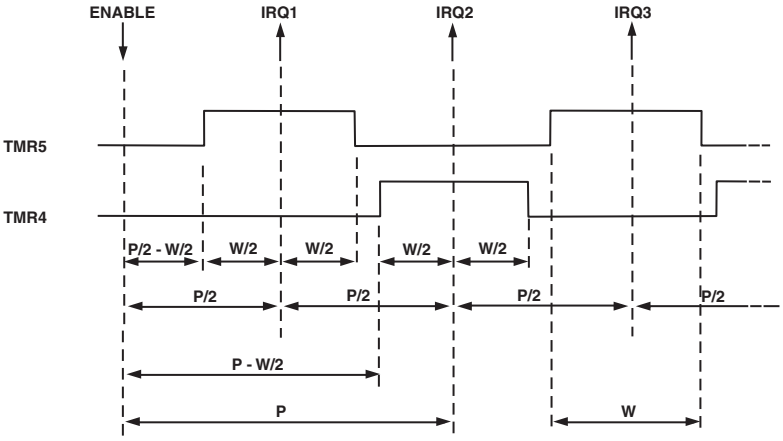


Figure 10-23. Non-Overlapping Clock Pulses

Since hardware only updates the written period and width values at the end of periods, software can write new values immediately after the timers have been enabled. Note that both timers' period expires at exactly the same times with the exception of the first timer 5 interrupt (at IRQ1) which is not visible to timer 4.

Listing 10-5. Non-Overlapping Clock Pulses

```
#define P 0x1000   /* signal period */
#define W 0x0600   /* signal pulse width */
#define N 4        /* number of pulses before disable */
timer45_toggle_hi:
    [--sp] = (r7:1, p5:5);
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
/* config timers */
    r7.l = IRQ_ENA | PERIOD_CNT | TOGGLE_HI | PULSE_HI | PWM_OUT;
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    r7.l = PERIOD_CNT | TOGGLE_HI | PULSE_HI | PWM_OUT;
    w[p5 + TIMER4_CONFIG - TIMER_ENABLE] = r7;
/* calculate timers widths and period */
    r0.l = lo(P);
    r0.h = hi(P);
    r1.l = lo(W);
    r1.h = hi(W);
    r2 = r1 >> 1;    /* W/2 */
    r3 = r0 >> 1;    /* P/2 */
    r4 = r3 - r2;    /* P/2 - W/2 */
    r5 = r0 - r2;    /* P - W/2 */
/* write values for initial period */
    [p5 + TIMER4_PERIOD - TIMER_ENABLE] = r0;
    [p5 + TIMER4_WIDTH - TIMER_ENABLE] = r5;
    [p5 + TIMER5_PERIOD - TIMER_ENABLE] = r3;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r4;
/* start timers */
    r7.l = TIMEN5 | TIMEN4 ;
    w[p5 + TIMER_ENABLE - TIMER_ENABLE] = r7;
/* write values for second period */
    [p5 + TIMER4_PERIOD - TIMER_ENABLE] = r3;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r2;
```

```
/* r0 functions as signal period counter */
    r0.h = hi(N * 2 - 1);
    r0.l = lo(N * 2 - 1);
    (r7:1, p5:5) = [sp++];
    rts;
timer45_toggle_hi.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:5, p5:5);
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
/* clear interrupt request */
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5 + TIMER_STATUS - TIMER_ENABLE] = r7;
/* toggle width values (width = period - width) */
    r7 = [p5 + TIMER5_PERIOD - TIMER_ENABLE];
    r6 = [p5 + TIMER5_WIDTH - TIMER_ENABLE];
    r5 = r7 - r6;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r5;
    r5 = [p5 + TIMER4_WIDTH - TIMER_ENABLE];
    r7 = r7 - r5;
    CC = r7 < 0;
    if CC r7 = r6;
    [p5 + TIMER4_WIDTH - TIMER_ENABLE] = r7;
/* disable after a certain number of periods */
    r0+= -1;
    CC = r0 == 0;
    r5.l = 0;
    r7.l = TIMDIS5 | TIMDIS4;
    if !CC r7 = r5;
    w[p5 + TIMER_DISABLE - TIMER_ENABLE] = r7;
    (r7:5, p5:5) = [sp++];
    astat = [sp++];
```

```
    rti;
isr_timer5.end:
```

Listing 10-5 generates N pulses on both timer output pins. Disabling the timers does not corrupt the generated pulse pattern anyhow.

Listing 10-6 configures timer 5 in WDTH_CAP mode. If looped back externally, this code might be used to receive N PWM patterns generated by one of the other timers. Ensure that the PWM generator and consumer both use the same PERIOD_CNT and PULSE_HI settings.

Listing 10-6. Timer Configured in WDTH_CAP Mode

```
.section L1_data_a;
.align 4;
#define N 1024
.var buffReceive[N*2];
.section L1_code;
timer5_capture:
    [--sp] = (r7:7, p5:5);
/* setup DAG2 */
    r7.h = hi(buffReceive);
    r7.l = lo(buffReceive);
    i2 = r7;
    b2 = r7;
    l2 = length(buffReceive)*4;
/* config timer for high pulses capture */
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
    r7.l = EMU_RUN|IRQ_ENA|PERIOD_CNT|PULSE_HI|WDTH_CAP;
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    r7.l = TIMEN5;
    w[p5 + TIMER_ENABLE - TIMER_ ENABLE] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
```

```
timer5_capture.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:7, p5:5);
/* clear interrupt request first */
    p5.h = hi(TIMER_STATUS);
    p5.l = lo(TIMER_STATUS);
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5] = r7;
    r7 = [p5 + TIMER5_PERIOD - TIMER_STATUS];
    [i2++] = r7;
    r7 = [p5 + TIMER5_WIDTH - TIMER_STATUS];
    [i2++] = r7;
    ssync;
    (r7:7, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```

Unique Behavior for the ADSP-BF52x Processor

The ADSP-BF52x processor features one general-purpose timer module that contains eight identical 32-bit timers. Each timer can be individually configured to operate in various modes. Although the timers operate completely independently of each other, all of them can be started and stopped simultaneously for synchronous operation.

Interface Overview

Figure 10-24 shows the ADSP-BF52x specific block diagram of the general-purpose timer module.

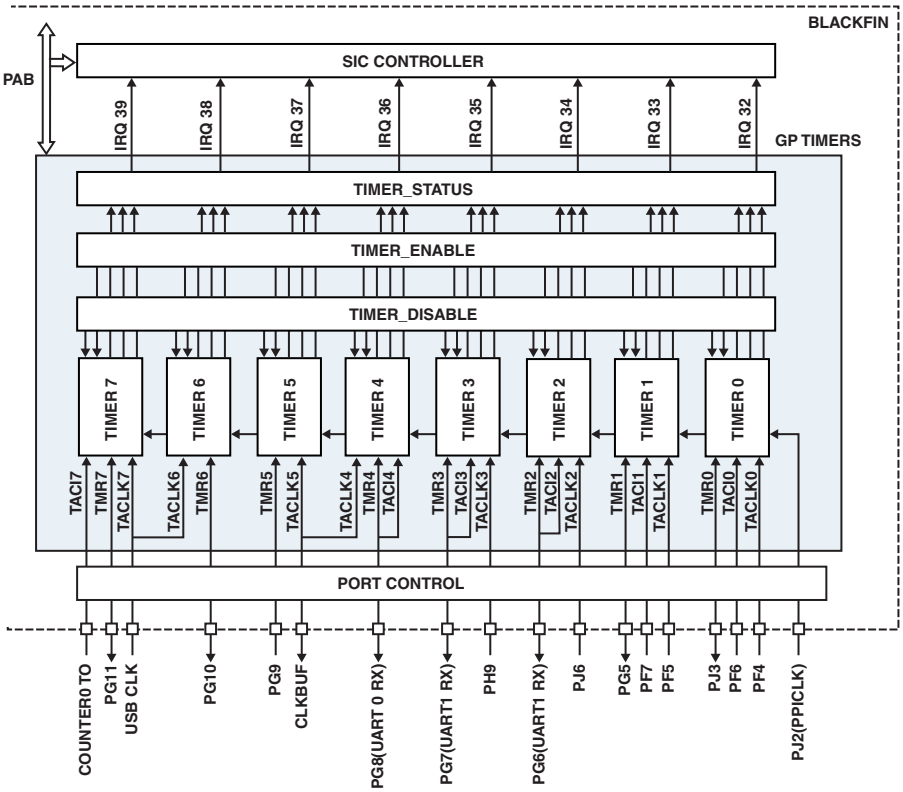


Figure 10-24. Timer Block Diagram

External Interface

The TMRCLK input is common to all eight timers. The PPI unit is clocked by the same pin; therefore any of the timers can be clocked by PPI_CLK.

Since timer 0 and timer 1 are often used in conjunction with the PPI, they are internally looped back to the PPI module for frame sync generation.

The timer signals TMR0 and TMR1 are multiplexed with the PPI frame syncs when the frame syncs are applied externally. PPI modes requiring only one frame sync free up TMR1. For details, see Chapter 15, "Parallel Peripheral Interface".

If the PPI frame syncs are applied externally, timer 0 and timer 1 are still fully functional and can be used for other purposes not involving the TMRx pins. Timer 0 and timer 1 must not drive their TMR0 and TMR1 pins. If operating in PWM_OUT mode, the OUT_DIS bit in the TIMER0_CONFIG and TIMER1_CONFIG registers must be set.

11 CORE TIMER

This chapter describes the core timer. Following an overview, functional description, and consolidated register definitions, the chapter concludes with a programming example.

Specific Information for the ADSP-BF52x

For details regarding the number of core timers for the ADSP-BF52x product, please refer to the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet*.

For Core Timer interrupt vector assignments, refer to Table 5-3 on page 5-18 in Chapter 5, "System Interrupts".

For a list of MMR addresses for each Core Timer, refer to Appendix A, "System MMR Assignments".

Core timer behavior for the ADSP-BF52x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Behavior for the ADSP-BF52x Processor" on page 11-9

Overview and Features

The core timer is a programmable 32-bit interval timer which can generate periodic interrupts. Unlike other peripherals, the core timer resides

inside the Blackfin core and runs at the core clock (CCLK) rate. Core timer features include:

- 32-bit timer with 8-bit prescaler
- Operates at core clock (CCLK) rate
- Dedicated high-priority interrupt channel
- Single-shot or continuous operation

Timer Overview

Figure 11-1 provides a block diagram of the core timer.

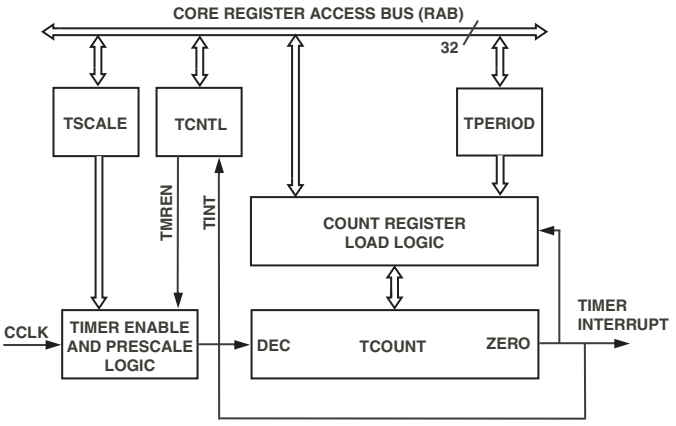


Figure 11-1. Core Timer Block Diagram

External Interfaces

The core timer does not directly interact with any pins of the chip.

Internal Interfaces

The core timer is accessed through the 32-bit register access bus (RAB). The module is clocked by the core clock CCLK. The timer's dedicated interrupt request is a higher priority than requests from all other peripherals.

Description of Operation

The software should initialize the TCOUNT register *before* the timer is enabled. The TCOUNT register can be written directly, but writes to the TPERIOD register are also passed through to TCOUNT.

When the timer is enabled by setting the TMREN bit in the core timer control register (TCNTL), the TCOUNT register is decremented once every time the prescaler TSCALE expires, that is, every TSCALE + 1 number of CCLK clock cycles. When the value of the TCOUNT register reaches 0, an interrupt is generated and the TINT bit is set in the TCNTL register.

If the TAUTORLD bit in the TCNTL register is set, then the TCOUNT register is reloaded with the contents of the TPERIOD register and the count begins again. If the TAUTORLD bit is not set, the timer stops operation.

The core timer can be put into low power mode by clearing the TMPWR bit in the TCNTL register. Before using the timer, set the TMPWR bit. This restores clocks to the timer unit. When TMPWR is set, the core timer may then be enabled by setting the TMREN bit in the TCNTL register.

Hardware behavior is undefined if TMREN is set when TMPWR = 0.

Interrupt Processing

The timer's dedicated interrupt request is a higher priority than requests from all other peripherals. The request goes directly to the core event controller (CEC) and does not pass through the system interrupt control-

ler (SIC). Therefore, the interrupt processing is also completely in the CCLK domain.

The core timer interrupt request is edge-sensitive and cleared by hardware automatically as soon as the interrupt is serviced.

The TINT bit in the TCNTL register indicates that an interrupt has been generated. Note that this is *not* a W1C bit. Write a 0 to clear it. However, the write is optional. It is not required to clear interrupt requests. The core time module doesn't provide any further interrupt enable bit. When the timer is enabled, interrupts can be masked in the CEC controller.

Core Timer Registers

The core timer includes four core memory-mapped registers, the timer control register (TCNTL), the timer count register (TCOUNT), the timer period register (TPERIOD), and the timer scale register (TSCALE). As with all core MMRs, these registers are always accessed by 32-bit read and write operations.

Core Timer Control Register (TCNTL)

The TCNTL register, shown in Figure 11-2, functions as control and status register.

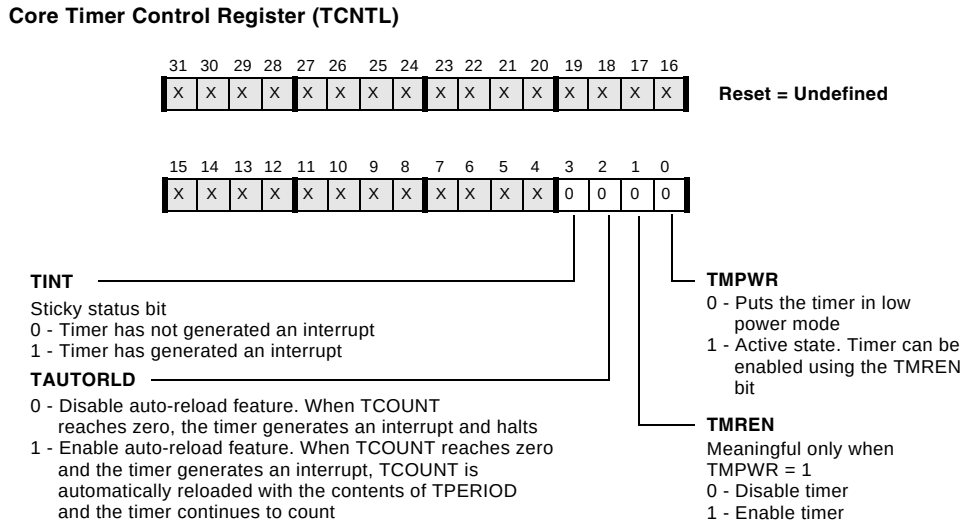

Core Timer Control Register (TCNTL)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X 0 0 0 0
TINT
Sticky status bit
0 - Timer has not generated an interrupt
1 - Timer has generated an interrupt
TAUTORLD
0 - Disable auto-reload feature. When TCOUNT reaches zero, the timer generates an interrupt and halts
1 - Enable auto-reload feature. When TCOUNT reaches zero and the timer generates an interrupt, TCOUNT is automatically reloaded with the contents of TPERIOD and the timer continues to count
TMPWR
0 - Puts the timer in low power mode
1 - Active state. Timer can be enabled using the TMREN bit
TMREN
Meaningful only when TMPWR = 1
0 - Disable timer
1 - Enable timer

Figure 11-2. Core Timer Control Register

Core Timer Count Register (TCOUNT)

The TCOUNT register, shown in Figure 11-3, decrements once every TSCALE + 1 clock cycles. When the value of TCOUNT reaches 0, an interrupt is generated and the TINT bit of the TCNTL register is set.

Values written to the TPERIOD register are automatically copied to the TCOUNT register. Nevertheless, the TCOUNT register can be written directly. In auto reload mode the value written to TCOUNT may differ from the TPERIOD value to let the initial period be shorter or longer than following periods. To do this, write to TPERIOD first and overwrite TCOUNT afterward.

Writes to TCOUNT are ignored once the timer is running.

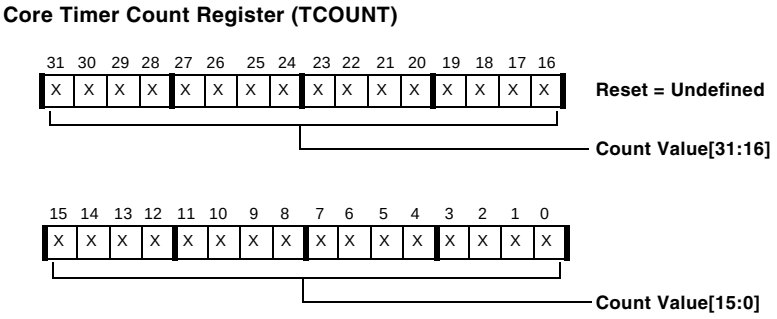

Core Timer Count Register (TCOUNT)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Count Value[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Count Value[15:0]

Figure 11-3. Core Timer Count Register

Core Timer Period Register (TPERIOD)

The TPERIOD register is shown in Figure 11-4. When auto-reload is enabled, the TCOUNT register is reloaded with the value of the TPERIOD register whenever TCOUNT reaches 0. Writes to TPERIOD are ignored when the timer is running.

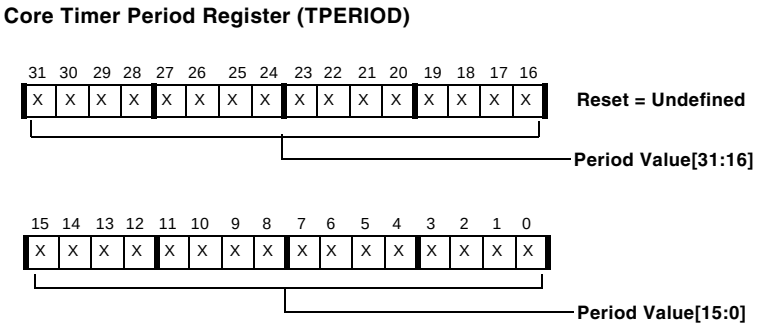

Core Timer Period Register (TPERIOD)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Period Value[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Period Value[15:0]

Figure 11-4. Core Timer Period Register

Core Timer Scale Register (TSCALE)

The TSCALE register is shown in Figure 11-5. The register stores the scaling value that is one less than the number of cycles between decrements of TCOUNT. For example, if the value in the TSCALE register is 0, the counter register decrements once every CCLK clock cycle. If TSCALE is 1, the counter decrements once every two cycles.

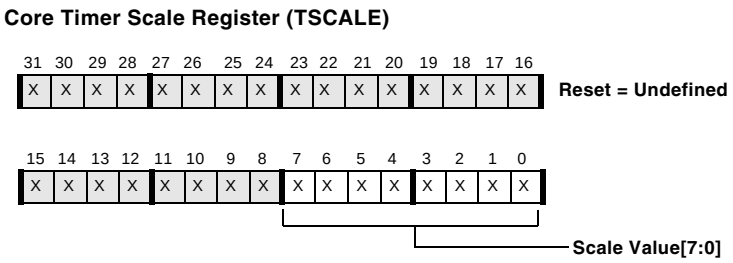

Core Timer Scale Register (TSCALE)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
x x x x x x x x x x x x x x x x
Reset = Undefined
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
x x x x x x x x x x x x x x x x
Scale Value[7:0]

Figure 11-5. Core Timer Scale Register

Programming Examples

Listing 11-1 configures the core timer in auto-reload mode. Assuming a CCLK of 500 MHz, the resulting period is 1 second. The initial period is twice as long as the others.

Listing 11-1. Core Timer Configuration

```
#include <defBF527.h>/*ADSP-BF527 product is used as an example*/
.section L1_code;
.global _main;
_main:
/* Register service routine at EVT6 and unmask interrupt */
    p1.l = lo(IMASK);
    p1.h = hi(IMASK);
```

```
    r0.l = lo(isr_core_timer);
    r0.h = hi(isr_core_timer);
    [p1 + EVT6 - IMASK] = r0;
    r0 = [p1];
    bitset(r0, bitpos(EVT_IVTMR));
    [p1] = r0;
/* Prescaler = 50, Period = 10,000,000, First Period = 20,000,000
*/
    p1.l = lo(TCNTL);
    p1.h = hi(TCNTL);
    r0 = 50 (z);
    [p1 + TSCALE - TCNTL] = r0;
    r0.l = lo(10000000);
    r0.h = hi(10000000);
    [p1 + TPERIOD - TCNTL] = r0;
    r0 <<= 1;
    [p1 + TCOUNT - TCNTL] = r0;
/* R6 counts interrupts */
    r6 = 0 (z);
/* start in auto-reload mode */
    r0 = TAUTORLD | TMPWR | TMREN (z);
    [p1] = r0;
_main.forever:
    jump _main.forever;
_main.end:
/* interrupt service routine simple increments R6 */
isr_core_timer:
    [--sp] = astat;
    r6+= 1;
    astat = [sp++];
    rti;
isr_core_timer.end:
```

Unique Behavior for the ADSP-BF52x Processor

None.

12 WATCHDOG TIMER

This chapter describes the watchdog timer. Following an overview, functional description, and consolidated register definitions, the chapter concludes with programming examples.

Specific Information for the ADSP-BF52x

For details regarding the number of watchdog timers for the ADSP-BF52x product, please refer to the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet*.

For Watchdog Timer interrupt vector assignments, refer to Table 5-3 on page 5-18 in Chapter 5, "System Interrupts".

For a list of MMR addresses for each Watchdog Timer, refer to Appendix A, "System MMR Assignments".

Watchdog timer behavior for the ADSP-BF52x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF52x Processor" on page 12-11

Overview and Features

The processor includes a 32-bit timer that can be used to implement a software watchdog function. A software watchdog can improve system reliability by generating an event to the processor core if the watchdog expires before being updated by software.

Watchdog timer key features include:

- 32-bit watchdog timer
- 8-bit disable bit pattern
- System reset on expire option
- NMI on expire option
- General-purpose interrupt option

Typically, the watchdog timer is used to supervise stability of the system software. When used in this way, software reloads the watchdog timer in a regular manner so that the downward counting timer never expires (never becomes 0). An expiring timer then indicates that system software might be out of control. At this point a special error handler may recover the system. For safety, however, it is often better to reset and reboot the system directly by hardware control.

Especially in slave boot configurations, a processor reset cannot automatically force the Blackfin device to be rebooted. In this case, the processor may reset without booting again and may negotiate with the host device by the time program execution starts. Alternatively, a watchdog event can cause an NMI event. The NMI service routine may request the host device reset and/or reboot the Blackfin processor.

The watchdog timer is often programmed to let the processor wake up from sleep mode after a programmable period of time.

For easier debugging, the watchdog timer does not decrement (even if enabled) when the processor is in emulation mode.

Interface Overview

Figure 12-1 provides a block diagram of the watchdog timer.

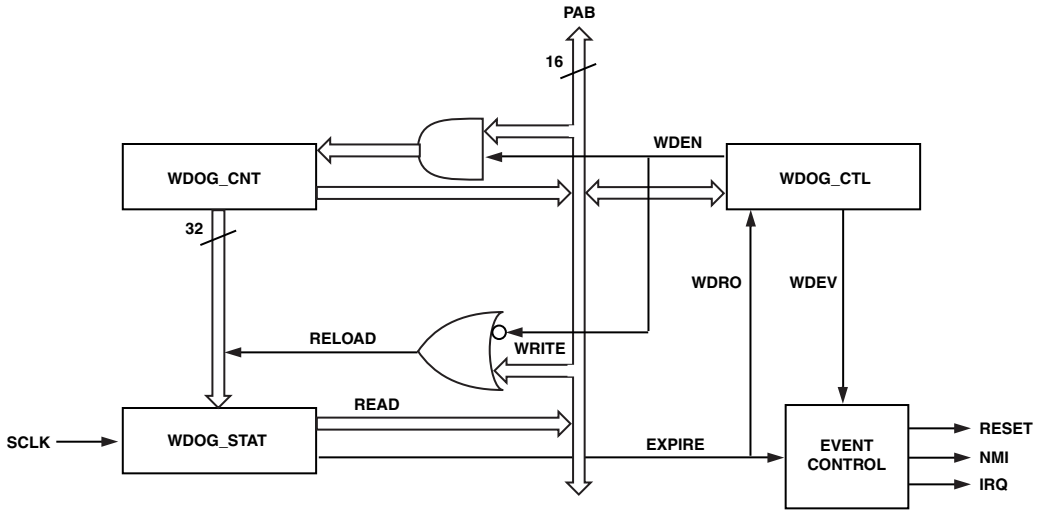


Figure 12-1. Watchdog Timer Block Diagram

External Interface

The watchdog timer does not directly interact with any pins of the chip.

Internal Interface

The watchdog timer is clocked by the system clock `SCLK`. Its registers are accessed through the 16-bit peripheral access bus (PAB). The 32-bit registers `WDOG_CNT` and `WDOG_STAT` must always be accessed by 32-bit read/write operations. Hardware ensures that those accesses are atomic.

When the counter expires, one of three event requests can be generated. Either a reset or an NMI request is issued to the core event controller

(CEC) or a general-purpose interrupt request is passed to the system interrupt controller (SIC).

Description of Operation

If enabled, the 32-bit watchdog timer counts downward every SCLK cycle. If it becomes 0, one of three event requests can be issued to either the CEC or the SIC. Depending on how the WDEV bit field in the WDOG_CTL register is programmed, the event that is generated may be a reset, a non-maskable interrupt, or a general-purpose interrupt.

The counter value can be read through the 32-bit WDOG_STAT register. The WDOG_STAT register cannot, however, be written directly. Rather, software writes the watchdog period value into the 32-bit WDOG_CNT register *before* the watchdog is enabled. Once the watchdog is started, the period value cannot be altered.

To start the watchdog timer:

1. Set the count value for the watchdog timer by writing the count value into the watchdog count register (WDOG_CNT). Since the watchdog timer is not enabled yet, the write to the WDOG_CNT registers automatically pre-loads the WDOG_STAT register as well.
2. In the watchdog control register (WDOG_CTL), select the event to be generated upon timeout.
3. Enable the watchdog timer in WDOG_CTL. The watchdog timer then begins counting down, decrementing the value in the WDOG_STAT register.

If software does not service the watchdog in time, WDOG_STAT continues decrementing until it reaches 0. Then, the programmed event is generated. The counter stops decrementing and remains at zero. Additionally, the WDRO latch bit in the WDOG_CTL register is set and can be interrogated by software in case event generation is not enabled.

When the watchdog is programmed to generate a reset, it resets the processor core and peripherals. If the NOBOOT bit in the SYSCR register was set by the time the watchdog resets the part, the chip is not rebooted. This is recommended behavior in slave boot configurations. The reset handler may evaluate the RESET_WDOG bit in the software reset register SWRST to detect a reset caused by the watchdog. For details, see the *System Reset and Booting* chapter.

To prevent the watchdog from expiring, software services the watchdog by performing dummy writes to the WDOG_STAT register. The values written are ignored, but the write commands cause the WDOG_STAT register to be reloaded from the WDOG_CNT register.

If the watchdog is enabled with a zero value loaded to the counter and the WDRO bit was cleared, the WDRO bit of the watchdog control register is set immediately and the counter remains at zero without further decrements. If, however, the WDRO bit was set by the time the watchdog is enabled, the counter decrements to 0xFFFF FFFF and continues operation.

Software can disable the watchdog timer only by writing a 0xAD value to the WDEN field in the WDOG_CTL register.

Register Definitions

The watchdog timer is controlled by three registers.

Watchdog Count (WDOG_CNT) Register

The WDOG_CNT register, shown in Figure 12-2, holds the 32-bit unsigned count value. The WDOG_CNT register must always be accessed with 32-bit read/writes.

A valid write to the WDOG_CNT register also preloads the watchdog counter. For added safety, the WDOG_CNT register can be updated only when the

watchdog timer is disabled. A write to the WDOG_CNT register while the timer is enabled does not modify the contents of this register.

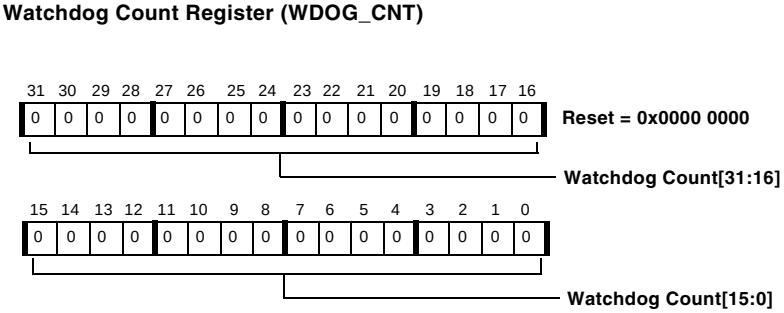


Figure 12-2. Watchdog Count Register

Watchdog Status (WDOG_STAT) Register

The 32-bit WDOG_STAT register, shown in Figure 12-3, contains the current count value of the watchdog timer. Reads to WDOG_STAT return the current count value. Values cannot be stored directly in WDOG_STAT, but are instead copied from WDOG_CNT. This can happen in two ways.

- While the watchdog timer is disabled, writing the WDOG_CNT register pre-loads the WDOG_STAT register.
- While the watchdog timer is enabled, but not rolled over yet, writes to the WDOG_STAT register load it with the value in WDOG_CNT.

Enabling the watchdog timer does not automatically reload WDOG_STAT from WDOG_CNT.

The WDOG_STAT register is a 32-bit unsigned system MMR that must be accessed with 32-bit reads and writes.

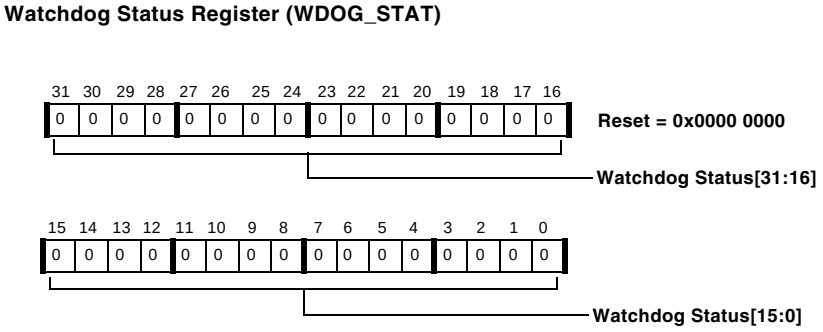


Figure 12-3. Watchdog Status Register

Watchdog Control (WDOG_CTL) Register

The WDOG_CTL register, shown in Figure 12-4, is a 16-bit system MMR used to control the watchdog timer.

The watchdog event (WDEV[1:0]) bit field is used to select the event that is generated when the watchdog timer expires. Note that if the general-purpose interrupt option is selected, the SIC_IMASK register that holds the watchdog timer mask bit should be appropriately configured to unmask that interrupt. If the generation of watchdog events is disabled, the watchdog timer operates as described, except that no event is generated when the watchdog timer expires.

The watchdog enable (WDEN[7:0]) bit field is used to enable and disable the watchdog timer. Writing any value other than the disable key (0xAD) into this field enables the watchdog timer. This multibit disable key minimizes the chance of inadvertently disabling the watchdog timer.

Software can determine whether the watchdog has expired by interrogating the WDRO status bit of the WDOG_CTL register. This is a sticky bit that is

set whenever the watchdog timer count reaches 0. It can be cleared only by writing a “1” to the bit when the watchdog has been disabled first.

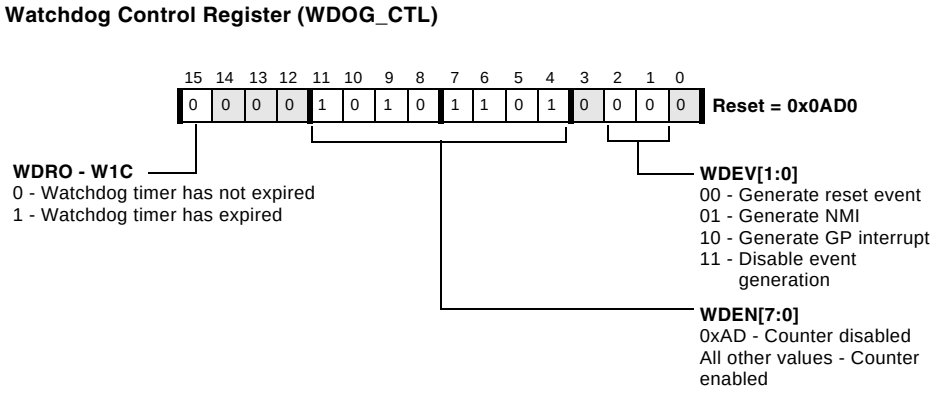

Watchdog Control Register (WDOG_CTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 1 0 1 1 0 1 0 0 0 0
Reset = 0x0AD0
WDRO - W1C
0 - Watchdog timer has not expired
1 - Watchdog timer has expired
WDEV[1:0]
00 - Generate reset event
01 - Generate NMI
10 - Generate GP interrupt
11 - Disable event generation
WDEN[7:0]
0xAD - Counter disabled
All other values - Counter enabled

Figure 12-4. Watchdog Control Register

Programming Examples

Listing 12-1 shows how to configure the watchdog timer so that it resets the chip when it expires. At startup, the code evaluates whether the recent reset event has been caused by the watchdog. Additionally, the example sets the NOBOOT bit to prevent the memory from being rebooted.

Listing 12-1. Watchdog Timer Configuration

```
#include <defBF527.h>/*ADSP-BF527 product is used as an example*/
#define WDOGPERIOD 0x00200000

.section L1_code;
.global _reset;
_reset:
  ...
/* optionally, test whether reset was caused by watchdog */
```

```
    p0.h=hi(SWRST);
    p0.l=lo(SWRST);
    r6 = w[p0] (z);
    CC = bittst(r6, bitpos(RESET_WDOG));
    if !CC jump _reset.no_watchdog_reset;

/* optionally, warn at system level or host device here */

_reset.no_watchdog_reset:
/* optionally, set NOBOOT bit to avoid reboot in case */
    p0.h=hi(SYSCR);
    p0.l=lo(SYSCR);
    r0 = w[p0](z);
    bitset(r0,bitpos(NOBOOT));
    w[p0] = r0;


/* start watchdog timer, reset if expires */
    p0.h = hi(WDOG_CNT);
    p0.l = lo(WDOG_CNT);
    r0.h = hi(WDOGPERIOD);
    r0.l = lo(WDOGPERIOD);
    [p0] = r0;
    p0.l = lo(WDOG_CTL);
    r0.l = WDEN | WDEV_RESET;
    w[p0] = r0;
    ...
    jump _main;
_reset.end:
```

The subroutine shown in Listing 12-2 can be called by software to service the watchdog. Note that the value written to the `WDOG_STAT` register does not matter.

Listing 12-2. Service Watchdog

```
service_watchdog:
    [--sp] = p5;
    p5.h = hi(WDOG_STAT);
    p5.l = lo(WDOG_STAT);
    [p5] = r0;
    p5 = [sp++];
    rts;
service_watchdog.end:
```

Listing 12-3 is an interrupt service routine that restarts the watchdog. Note that the watchdog must be disabled first.

Listing 12-3. Watchdog Restarted by Interrupt Service Routine

```
isr_watchdog:
    [--sp] = astat;
    [--sp] = (p5:5, r7:7);
    p5.h = hi(WDOG_CTL);
    p5.l = lo(WDOG_CTL);
    r7.l = WDDIS;
    w[p5] = r7;
    bitset(r7, bitpos(WDRO));
    w[p5] = r7;
    r7 = [p5 + WDOG_CNT - WDOG_CTL];
    [p5 + WDOG_CNT - WDOG_CTL] = r7;
    r7.l = WDEN | WDEV_GPI;
    w[p5] = r7;
    (p5:5, r7:7) = [sp++];
    astat = [sp++];
    rti;
isr_watchdog.end:
```

Unique Information for the ADSP-BF52x Processor

None.

13 GENERAL-PURPOSE COUNTER

This chapter describes the general-purpose up/down counter. The counter provides support for manually controlled rotary controllers, such as the volume wheel on a radio device. This unit also supports industrial encoders. Following the overview and list of key features is a description of the operating modes.

This chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF52x

For details regarding the number of GP counters for the ADSP-BF52x product, please refer to the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet*.

For GP counter interrupt vector assignments, refer to Table 5-3 on page 5-18 in Chapter 5, "System Interrupts".

To determine how each of the GP counters is multiplexed with other functional pins, refer to Table 9-2 on page 9-5 through Table 9-5 on page 9-9 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for each GP counter, refer to Appendix A, "System MMR Assignments".

GP counter behavior for the ADSP-BF52x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Behavior for the ADSP-BF52x Processor" on page 13-38

Overview

The purpose of this interface is to convert pulses from incremental position encoders into data that is representative of the actual position. This is done by integrating (counting) pulses on one or two inputs. Since integration provides relative position, some devices also feature a zero position input (zero marker) that can be used to establish a reference point to verify that the acquired position does not drift over time.

In addition, the incremental position information can be used to determine speed, if the time intervals are measured.

The GP counter provides flexible ways to establish position information. When used in conjunction with the GP timer block, the GP counter allows for the acquisition of coherent position/time-stamp information that enables speed calculation.

Features

The GP counter includes the following features:

- 32-bit up/down counter
- Quadrature encoder mode (Gray code)
- Binary encoder mode
- Alternative frequency-direction mode
- Timed direction and up/down counting modes
- Zero marker/push button support
- Capture event timing in association with general purpose timer
- Boundary comparison and boundary setting features

- Input pin noise filtering (debouncing)
- Flexible error detection/signaling

Interface Overview

A block diagram of the GP counter is shown in Figure 13-1. There are two input pins, the count up and direction (CUD) pin and the count down and gate (CDG) pin, that accept various forms of incremental inputs and are processed by the 32-bit counter. The third input, count zero marker (CZM), is the zero marker input. The module interfaces to the processor by way of the peripheral access bus (PAB) and can optionally generate an interrupt request through the IRQ line. There is also an output that can be used by the timer module to generate time-stamps on certain events.

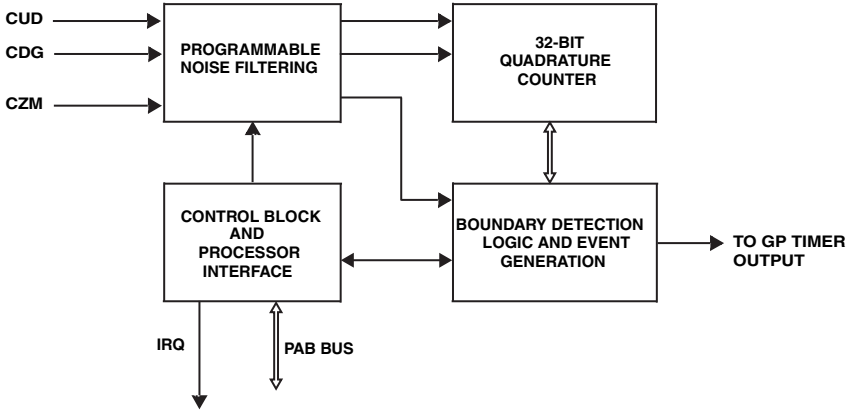


Figure 13-1. Block Diagram of the GP Counter Interface

Description of Operation

The GP counter has five modes of operation that are described in this section.

With the exception of the timed direction mode, the GP counter can operate with the GP timer block to capture additional timing information (time-stamps) associated with events detected by this block.

The third input (CZM) may be used as a zero marker or to sense the pressing of a push button. Refer to "Zero Marker (Push Button) Operation" on page 13-9 for more details.

The three input pins may be filtered (debounced) before being evaluated by the GP counter. Refer to "Input Noise Filtering (Debouncing)" on page 13-8 for more details.

The GP counter also features a flexible boundary comparison. In all of the operating modes, the counter can be compared to an upper and lower limit. A variety of actions can be taken when these limits are reached. Refer to "Boundary Comparison Modes" on page 13-10 for more details.

Quadrature Encoder Mode

In this mode, the CUD:CDG inputs expect a quadrature-encoded signal that is interpreted as a 2-bit Gray code. The order of transitions of the CUD and CDG inputs determines whether the counter increments or decrements. The CNT_COUNTER register contains the number of transitions that have occurred. Refer to Table 13-1 for more details.

Optionally, an interrupt is generated if both inputs change within one SCLK cycle. Such transitions are not allowed by Gray coding. Therefore,

the CNT_COUNTER register remains unchanged and an error condition is signaled.

Table 13-1. Quadrature Events and Counting Mechanism

CNT_COUNTER Register Value	–4	–3	–2	–1	0	+1	+2	+3	+4
CDG:CUD Inputs	00	01	11	10	00	01	11	10	00

It is possible to reverse the count direction of the Gray coded signal. This can be achieved by enabling the polarity inverter of either the CUD pin or the CDG pin. Inverting both pins will not alter the behavior. This feature can be enabled with the CDGINV and CUDINV bits in the CNT_CONFIG register.

As an example, if the CDG:CUD inputs are 00 respectively and the next transition is to 01, this would normally increment the counter as is shown in Table 13-1 on page 13-5. If the CUD polarity is inverted this generates a received input of 01 followed by 00. This will result in a decrement of the counter, altering the behavior of the connected hardware.

Binary Encoder Mode

This mode is almost identical to the previous mode, with the exception that the CUD:CDG inputs expect a binary-encoded signal. The order of transitions of the CUD and CDG inputs determines whether the counter increments or decrements. The CNT_COUNTER register contains the number of transitions that have occurred. Refer to Table 13-2.

Optionally, an interrupt is generated if the detected code steps by more than 1 (in binary arithmetic) within one SCLK cycle. Such transitions are

considered erroneous. Therefore, the CNT_COUNTER register remains unchanged and an error condition is signaled.

Table 13-2. Binary Events and Counting Mechanism

CNT_COUNTER Register Value	–4	–3	–2	–1	0	+1	+2	+3	+4
CDG:CUD Inputs	00	01	10	11	00	01	10	11	00

Reversing the CUD and CDG pin polarity has a different effect for the binary encoder mode than for the quadrature encoder mode. Inverting the polarity of the CUD pin only, or inverting both the CUD and CDG pins, will result in reversing the count direction.

Up/Down Counter Mode

In this mode, the counter is incremented or decremented at every active edge of the input pins.

If an active edge is detected at the CUD input, the counter increments. The active edge can be selected by the CUDINV bit in the CNT_CONFIG register. If this bit is cleared, a rising edge will increment the counter. If this bit is set, a falling edge will increment the counter.

If an active edge is detected at the CDG input, the counter decrements. The active edge can be selected by the CDGINV bit in the CNT_CONFIG register. If this bit is cleared, a rising edge will decrement the counter. If this bit is set, a falling edge will decrement the counter.

If simultaneous edges occur on pin CDG and pin CUD, the counter remains unchanged and both up-count and down-count events are signaled in the CNT_STATUS register.

Direction Counter Mode

In this mode, the counter is incremented or decremented at every active edge of the CDG input pin.

The state of the CUD input determines whether the counter increments or decrements. The polarity can be selected by the CUDINV bit in the CNT_CONFIG register. If this bit is cleared, a high CUD input will increment, a low input will decrement. If this bit is set, the polarity is inverted.

If an active edge is detected at the CDG input, the counter value changes by one in the selected direction. The active edge can be selected by the CDGINV bit in the CNT_CONFIG register. If this bit is cleared, a rising edge will decrement the counter. If this bit is set, a falling edge will decrement the counter.

Timed Direction Mode

In this mode, the counter is incremented or decremented at each SCLK cycle.

The state of the CUD input determines whether the counter increments or decrements. The polarity can be selected by the CUDINV bit in the CNT_CONFIG register. If this bit is cleared, a high CUD input will increment the counter, a low input will decrement it. If this bit is set, the polarity is inverted.

The CDG pin can be used to gate the clock. The polarity can be selected by the CDGINV bit in the CNT_CONFIG register. If this bit is cleared, a high CDG input will enable the counter, a low input will stop it. If this bit is set, the polarity is inverted.

Functional Description

The following sections describe the various functions in more detail.

Input Noise Filtering (Debouncing)

In all modes, the three input pins can be filtered to present clean signals to the GP counter logic. This filtering can be enabled or disabled by the DEBE bit in the CNT_CONFIG register. Figure 13-2 shows the filtering operation for the CUD pin.

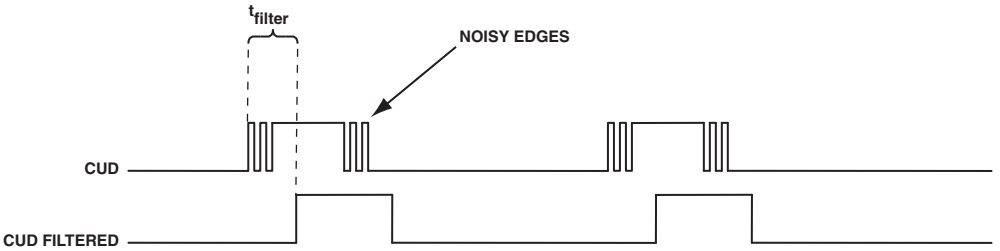


Figure 13-2. Programmable Noise Filtering

The filtering mechanism is implemented using counters for each pin. The counter for each pin is initialized from the DPRESCALE field of the CNT_DEBOUNCE register. When a transition is detected on a pin, the corresponding counter starts counting up to the programmed number of SCLK cycles. The state of the pin is latched after time t_{filter} and passed on to the GP counter logic.

The 5-bit DPRESCALE field in the CNT_DEBOUNCE register programs the desired number of cycles and therefore the debouncing time. The number of SCLK cycles for each pin can be selected in 18 steps ranging from 1 × 128 SCLK periods to 131072 × 128 SCLK periods (see Figure 13-9 on page 13-25).

The time t_{filter} is determined, given SCLK and the DPRESCALE value contained in the CNT_DEBOUNCE register, by the following formula:

$$t_{filter} = 128 \times (2^{DPRESCALE} \div SCLK)$$

where DPRESCALE can contain values from 0 (minimum filtering) to 17 (maximum filtering).

Assuming an SCLK frequency of 133 MHz, the filter time range is shown by the following equations:

DPRESCALE = 0b0000

$$t_{filter} = 128 * 1 * 7.5ns = 960ns = \textit{(approx.) } 1\mu s$$

DPRESCALE = 0b10001

$$t_{filter} = 128*(131072) * 7.5ns = 125829\mu s = \textit{(approx.) } 126ms$$

Zero Marker (Push Button) Operation

The CZM input pin can be used to sense the zero marker output of a rotary device or to detect the pressing of a push button. There are four programming schemes which are functional in all counter modes:

- **Push button mode**–This mode is enabled by setting the CZMIE bit in the CNT_IMASK register. An active edge at the CZM input will set the CZMII bit in the CNT_STATUS register. If enabled at the system interrupt controller, this will generate an interrupt request. The active edge is selected by the CZMINV bit in the CNT_CONFIG register (rising edge if cleared, falling edge if set to one).
- **Zero-marker-zeros-counter mode**–This mode is enabled by setting the ZMZC bit in the CNT_CONFIG register. An active level at the CZM input clears the CNT_COUNTER register and holds it until the CZM pin is deactivated. In addition, if enabled by the CZMZIE bit in the CNT_IMASK register, it will set the CZMZII bit in the CNT_STATUS register. If enabled by the peripheral interrupt controller, this will generate an interrupt request. The active level is selected by the CZMINV bit in the CNT_CONFIG register (active high if cleared, active low if set to one).

- **Zero-marker-error mode**–This mode is used to detect discrepancies between counter value and the zero marker output of certain rotary encoder devices. It is enabled by setting the CZMEIE bit in the CNT_IMASK register. When an active edge is detected at the CZM input pin, the four LSBs of the CNT_COUNTER register are compared to zero. If they are not zero, a mismatch is signaled by way of the CZMEII bit in the CNT_STATUS register. If enabled by the peripheral interrupt controller, this will generate an interrupt request. The active edge is selected by the CZMINV bit in the CNT_CONFIG register: (rising edge if cleared, falling edge if set to one).

- **Zero-once mode**–This mode is used to perform an initial reset of the counter value when an active zero marker is detected. After that, the zero marker is ignored (the counter is not reset anymore). This mode is enabled by setting the W1ZMONCE bit in the CNT_COMMAND register. The CNT_COUNTER register and the W1ZMONCE bit are cleared on the next active edge on the CZM pin. Thus, the W1ZMONCE bit can be read to check whether the event has already occurred, if desired. The active edge of the CZM pin is selected by the CZMINV bit in the CNT_CONFIG register (rising edge if cleared, falling edge if set to one).

Boundary Comparison Modes

The GP counter includes two boundary registers, CNT_MIN (lower) and CNT_MAX (upper). The counter value is compared to the lower and upper boundary. Depending on which mode is selected, different actions are taken if the count value reaches either of the boundary values.

There are four boundary modes:

- **Boundary-compare mode**–The two boundary registers are simply compared to the CNT_COUNTER register. If, after incrementing, CNT_COUNTER equals CNT_MAX then the MAXCII bit in the CNT_STATUS register is set. If the MAXCIE bit in the CNT_IMASK register is set, an

interrupt request is generated. Similarly if, after decrementing, CNT_COUNTER equals CNT_MIN then the MINCII status bit is set. If the MINCIE bit in the CNT_IMASK register is set, an interrupt request is generated. The MAXCII and MINCII bits are not set if the CNT_MAX and CNT_MIN registers are updated by software.

- **Boundary-zero mode**–This mode is similar to the boundary-compare mode. In addition to setting the status bits and requesting interrupts, the counter value in the CNT_COUNTER register is also set to zero.

- **Boundary auto-extend mode**–In this mode, the boundary registers are modified by hardware whenever the counter value reaches either of them. The CNT_MAX register is loaded with the current CNT_COUNTER value if the latter increments beyond the CNT_MAX value. Similarly, the CNT_MIN register is loaded with the CNT_COUNTER value if the latter decrements below the CNT_MIN value. This mode may be used to keep track of the widest angle the wheel ever reported, even if the software did not serve interrupts. At startup, the application software should set both boundary registers to the initial CNT_COUNTER value. The MAXCII and MINCII status bits are still set when the counter value matches the boundary register.

- **Boundary-capture mode**–In this mode, the CNT_COUNTER value is latched into the CNT_MIN register at one detected edge of the CZM input pin, and latched into CNT_MAX at the opposite edge. If the CZMINV bit in the CNT_CONFIG register is cleared, a rising edge captures into CNT_MIN and a falling edge into CNT_MAX. If the CZMINV bit is set, the edges are inverted. The MAXCII and MINCII status bits report the capture event.

The comparison is performed with signed arithmetic. The boundary registers and the counter value are all treated as signed integer values.

Control and Signaling Events

Eleven events can be signaled to the processor using status information and optional interrupt requests. The interrupts are enabled by the respective bits in the CNT_IMASK register. Dedicated bits in the CNT_STATUS register report events. When an interrupt from the GP counter is acknowledged, the application software is responsible for correct interpretation of the events. It is recommended to logically AND the content of the CNT_IMASK and CNT_STATUS registers to identify pending interrupts. Interrupt requests are cleared by write-one-to-clear (W1C) operations to the CNT_STATUS register. Hardware does not clear the status bits automatically, unless the counter module is disabled.

Illegal Gray/Binary Code Events

When the illegal transitions described in "Quadrature Encoder Mode" on page 13-4 or "Binary Encoder Mode" on page 13-5 occur, the ICII bit in the CNT_STATUS register is set. If enabled by the ICIE bit in the CNT_IMASK register, an interrupt request is generated. The ICIE bit should only be set in the quadrature encoder or binary encoder modes.

Up/Down Count Events

The UCII bit in the CNT_STATUS register indicates whether the counter has been incremented. Similarly, the DCII bit reports decrements. The two events are independent. For instance, if the counter first increments by one and then decrements by two, both bits remain set, even though the resulting counter value shows a decrement by one. In up/down counter mode, hardware may detect simultaneous active edges on the CUD and CDG inputs. In that case, the CNT_COUNTER remains unchanged, but both the UCII and DCII bits are set.

Interrupt requests for these events may be enabled through the UCIE and DCIE bits. This feature should be used carefully when the counter is

clocked at high rates. This is especially critical when the counter operates in DIR_TMR mode, as interrupts would be generated every SCLK cycle.

These events can also be used for additional push buttons, if GP counter features are not needed. When up/down counter mode is enabled, these count events can be used to report interrupts from push buttons that connect to the CUD and CDG inputs.

Zero-Count Events

The CZEROII status bit indicates that the CNT_COUNTER has reached a value equal to 0x0000 0000 after an increment or decrement. This bit is not set when the counter value is set to zero by a write to CNT_COUNTER or by setting the W1LCNT_ZERO bit in the CNT_COMMAND register. If enabled by the CZEROIE bit, an interrupt request is generated.

Overflow Events

There are two status bits that indicate whether the signed counter register has overflowed from a positive to a negative value or vice versa.

The COV31II bit reports that the 32-bit CNT_COUNT register has either incremented from 0x7FFF FFFF to 0x8000 0000, or decremented from 0x8000 0000 to 0x7FFF FFFF. If enabled by the COV31IE bit, an interrupt request is generated.

Similarly, in applications where only the lower 16 bits of the counter are of interest, the COV15II status bit reports counter transitions from 0xXXXX 7FFF to 0xXXXX 8000, or from 0xXXXX 8000 to 0xXXXX 7FFF. If enabled by the COV15IE bit, an interrupt request is generated.

Boundary Match Events

The MINCII and MAXCII status bits report boundary events as described in "Boundary Comparison Modes" on page 13-10. These bits are not set if

the CNT_COUNTER, CNT_MAX or CNT_MIN registers are updated by software or the CNT_COMMAND register is written to.

The MINCIE and MAXCIE bits in the CNT_IMASK register enable interrupt generation on boundary events.

Zero Marker Events

There are three status bits CZMII, CZMEII and CZMZII associated with zero marker events, as described in "Zero Marker (Push Button) Operation" on page 13-9. Each of these events can optionally generate an interrupt request, if enabled by the corresponding CZMIE, CZMEIE and CZMZIE bits in the CNT_IMASK register.

Capturing Timing Information

To calculate speed, many applications may wish to measure the time between two count events—in addition to accurately counting encoder pulses. For more accuracy, particularly at very low speeds, it is also necessary to obtain the time that has elapsed since the last count event. This additional information allows for estimating how much the GP counter has advanced since the last counter event.

For this purpose, the GP counter has an internal signal that connects to the alternate capture input (TACIx) of one of the GP timers. It is functional in all modes, with the exception of the timed direction mode. Refer to the "Internal Interfaces" section of Chapter 9, "General-Purpose Ports" for information regarding which GP timer(s) are associated with which GP counter module(s) for your device.

In order to use the timing measurements, the associated GP timer must be used in the WDTH_CAP mode. The alternate capture input is selected by setting the TIN_SEL bit in the GP timer's TIMER_CONFIG register. For more information about the GP timers and their operating modes, refer to the *General-Purpose Timer* chapter.

Capturing Time Interval Between Successive Counter Events

When the only timing information of interest is the interval between successive count events, the associated timer should be programmed in WDTH_CAP mode with PULSE_HI = 1, PERIOD_CNT = 1 and TIN_SEL = 1. Typically, this information is sufficient if the speed of GP counter events is known not to reach very low values. Figure 13-3 shows the operation of the GP counter and the GP timer in this mode. TO generates a pulse every time a count event occurs. The GP timer will update its TIMER_PERIOD register with the period (measured from rising edge to rising edge) of the TO signal. The TIMER_PERIOD register is updated at every rising edge of the TO signal and contains the number of system clock (SCLK) cycles that have elapsed since the previous rising edge.

Incidentally, the TIMER_WIDTH register is also updated at the same time, but is generally of no interest in this mode of operation. If no reads of the CNT_COUNTER register occur between counter events, the TIMER_WIDTH register only contains the width of the TO pulse. If a read of CNT_COUNTER has occurred between events, the TIMER_WIDTH register will contain the time between the read of CNT_COUNTER and the next event.

This mode can also be used with PULSE_HI = 0. In this case, the period of TO is measured between falling edges. It will result in the same values as in the previous case, only the latching occurs one SCLK cycle later.

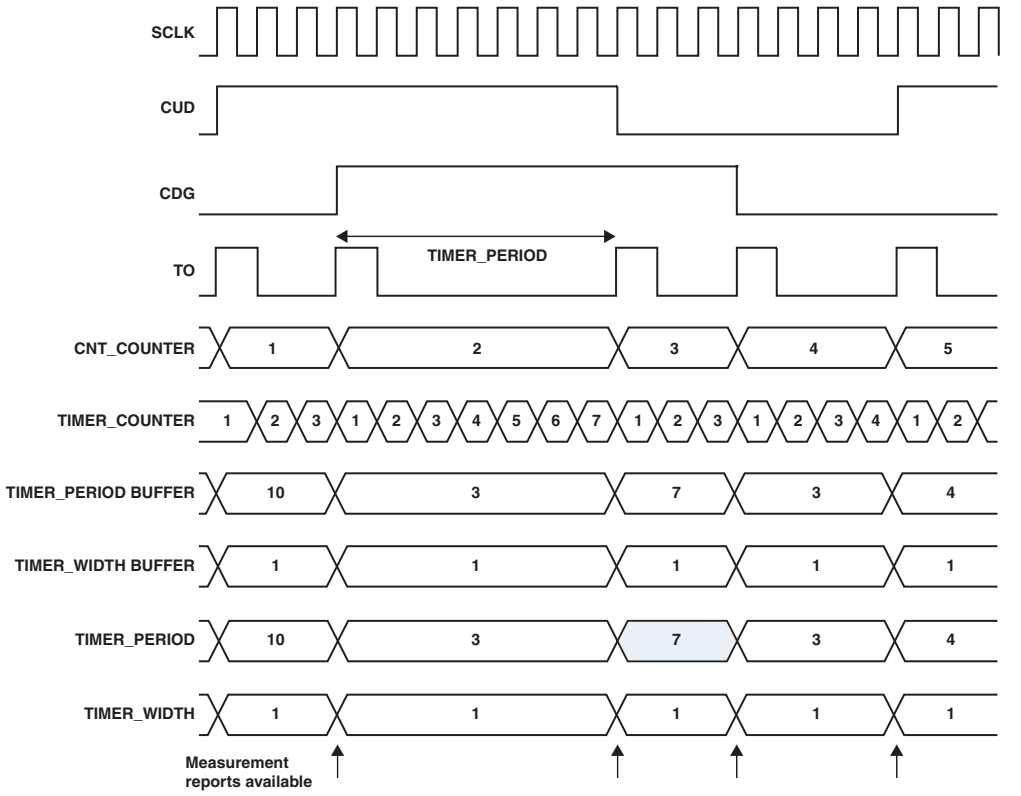


Figure 13-3. Operation with GP Timer Module

Capturing Counter Interval and CNT_COUNTER Read Timing

It is possible to also capture the time elapsed since the last count event. In this mode, the associated timer should be programmed in WDTH_CAP mode with PULSE_HI = 0, PERIOD_CNT = 0 and TIN_SEL = 1. Typically, this additional information is used to estimate the advancement of the GP counter

since the last count event, when the speed is very low. Figure 13-4 shows the operation of the GP counter module and the GP timer module in this mode. TO generates a pulse every time a count event occurs. In addition, when the processor reads the CNT_COUNTER register, the TO signal presents a pulse which is extended (high) until the next count event. The GP timer will update its TIMER_PERIOD register with the period (measured from falling edge to falling edge, because PULSE_HI = 0) of the TO signal. The TIMER_WIDTH register is updated with the pulse width (the portion where TO is low, again because PULSE_HI = 0). Both registers are updated at every rising edge of the TO signal (because PERIOD_CNT = 0). Therefore, the TIMER_PERIOD register contains the period between the last two count events and the TIMER_WIDTH register contains the time since the last count event and the read of the CNT_COUNTER register, both measured in number of SCLK cycles.

The result is that when reading the CNT_COUNTER register, the two time measurements are also latched and the user has a coherent triplet of information to calculate speed and position.

Restrictions apply to the use of the TO signal in terms of speed. Therefore, the user must take care to not operate at very high count events. For instance, if CNT_COUNTER is incremented/decremented every SCLK cycle (timed direction mode), the TO signal is incorrect.

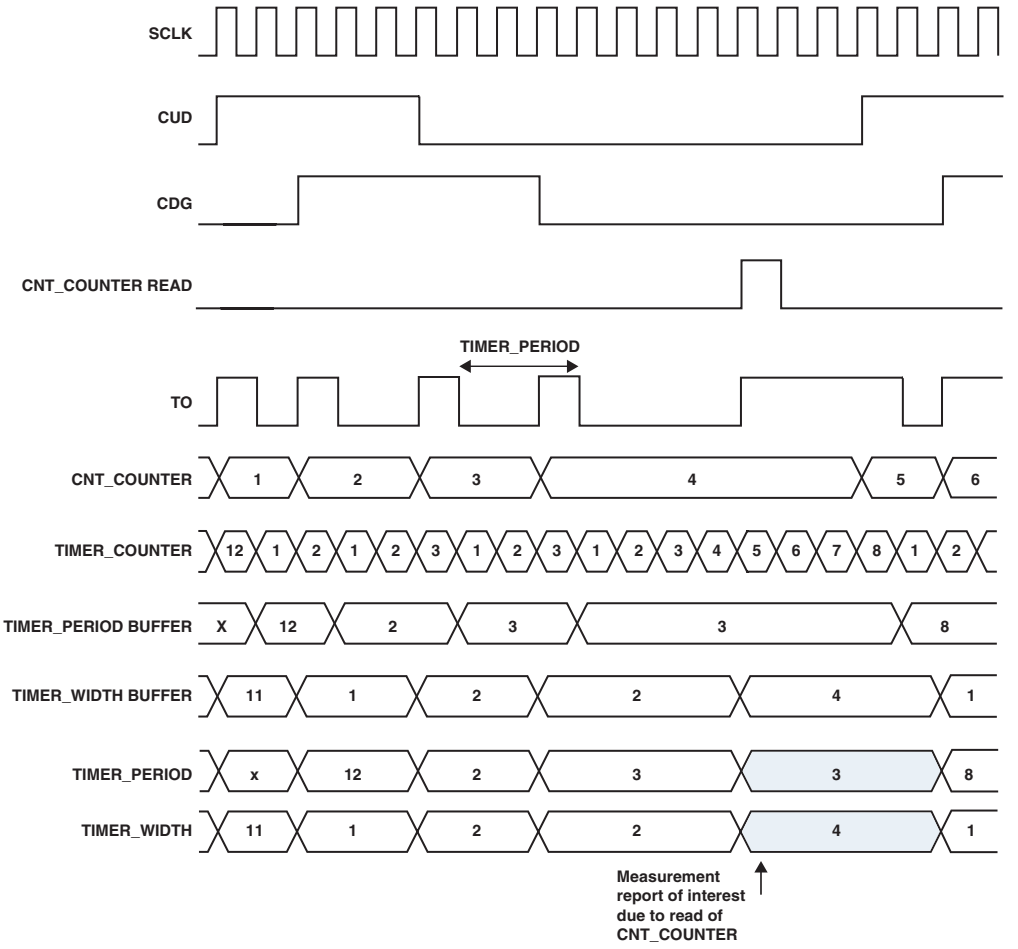


Figure 13-4. Capturing Counter Interval

Programming Model

In a typical application, the user will initialize the GP counter for the desired mode, without enabling it. Normally the events of interest will be processed using interrupts rather than polling the status bit. In that case,

clear all status bits and activate the generation of interrupt requests with the CNT_IMASK register. Set up the system interrupt controller and core interrupts. If timing information is required, set up the relevant GP timer in WDTH_CAP mode with the settings described in the "Capturing Timing Information" on page 13-14. Then, enable the interrupts and the peripheral itself.

Registers

The GP counter interface has eight memory-mapped registers (MMRs) that regulate its operation. Descriptions and bit diagrams for MMRs is provided in the sections that follow.

Counter Module Register Overview

Refer to Table 13-3 for an overview of all MMRs associated with the GP counter interface.

Table 13-3. Counter Module Register Overview

Register Name	Width	PAB Operation	Reset Value
CNT_CONFIG	16 bits	R/W	0x0000
CNT_IMASK	16 bits	R/W	0x0000
CNT_STATUS	16 bits	R/W1C	0x0000
CNT_COMMAND	16 bits	R/W1A	0x0000
CNT_DEBOUNCE	16 bits	R/W	0x0000
CNT_COUNTER	32 bits	R/W (16/32 bits)	0x0000 0000
CNT_MAX	32 bits	R/W (16/32 bits)	0x0000 0000
CNT_MIN	32 bits	R/W (16/32 bits)	0x0000 0000

Counter Configuration Register (CNT_CONFIG)

This register is used to configure counter modes and input pins, as well as to enable the peripheral. It can be accessed at any time with 16-bit read and write operations.

Counter Configuration (CNT_CONFIG) Register

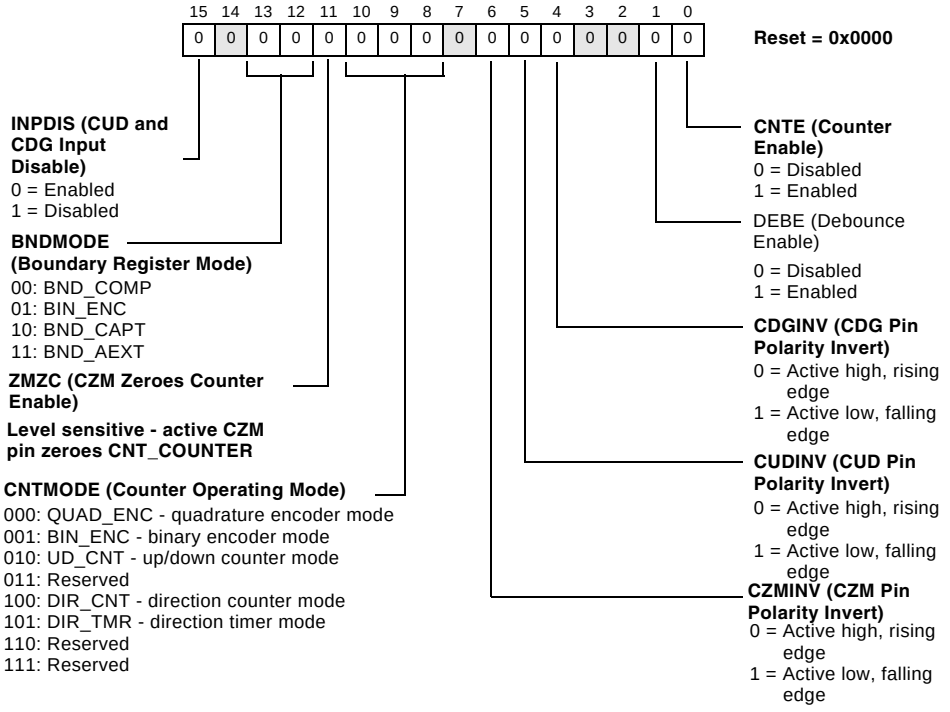

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
INPDIS (CUD and CDG Input Disable)
0 = Enabled
1 = Disabled
BNDMODE (Boundary Register Mode)
00: BND_COMP
01: BIN_ENC
10: BND_CAPT
11: BND_AEXT
ZMZC (CZM Zeroes Counter Enable)
Level sensitive - active CZM pin zeroes CNT_COUNTER
CNTMODE (Counter Operating Mode)
000: QUAD_ENC - quadrature encoder mode
001: BIN_ENC - binary encoder mode
010: UD_CNT - up/down counter mode
011: Reserved
100: DIR_CNT - direction counter mode
101: DIR_TMR - direction timer mode
110: Reserved
111: Reserved
CNTE (Counter Enable)
0 = Disabled
1 = Enabled
DEBE (Debounce Enable)
0 = Disabled
1 = Enabled
CDGINV (CDG Pin Polarity Invert)
0 = Active high, rising edge
1 = Active low, falling edge
CUDINV (CUD Pin Polarity Invert)
0 = Active high, rising edge
1 = Active low, falling edge
CZMINV (CZM Pin Polarity Invert)
0 = Active high, rising edge
1 = Active low, falling edge

Figure 13-5. Counter Configuration Register

Counter Interrupt Mask Register (CNT_IMASK)

This register is used to enable interrupt request generation from each of the eleven events. It can be accessed at any time with 16-bit read and write

operations. For explanations of the register bits, refer to the “Control and Signaling Events” on page 13-12.

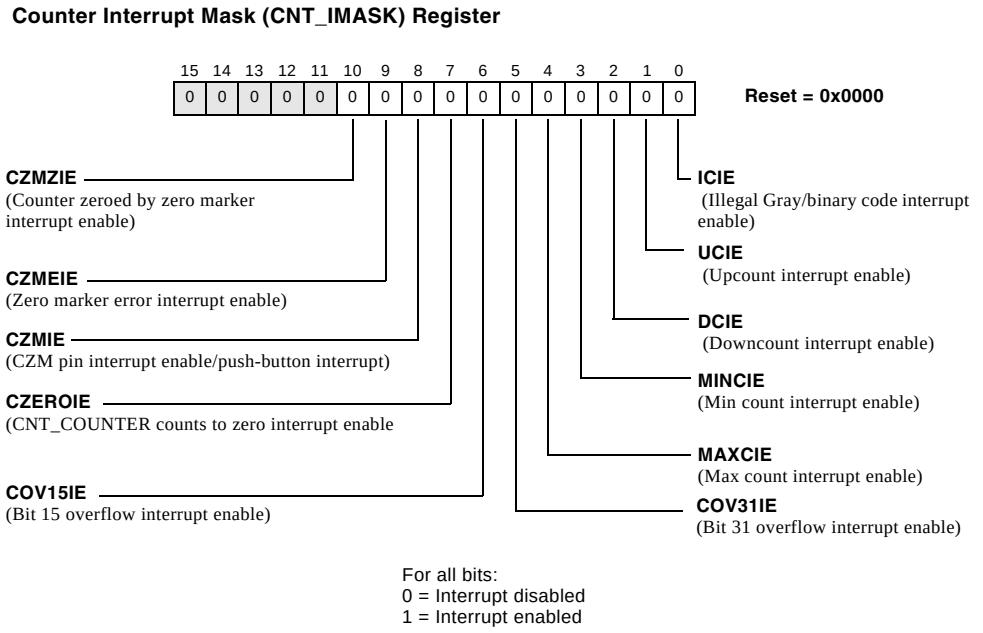


Figure 13-6. Counter Interrupt Mask Register

Counter Status Register (CNT_STATUS)

This register provides status information for each of the eleven events where 0 = no interrupt pending and 1 = interrupt pending. When an event is detected, the corresponding bit in this register is set. It remains set until either software writes a “1” to the bit (write-1-to-clear) or the GP counter

is disabled. For explanations of the register bits, refer to the “Control and Signaling Events” on page 13-12.

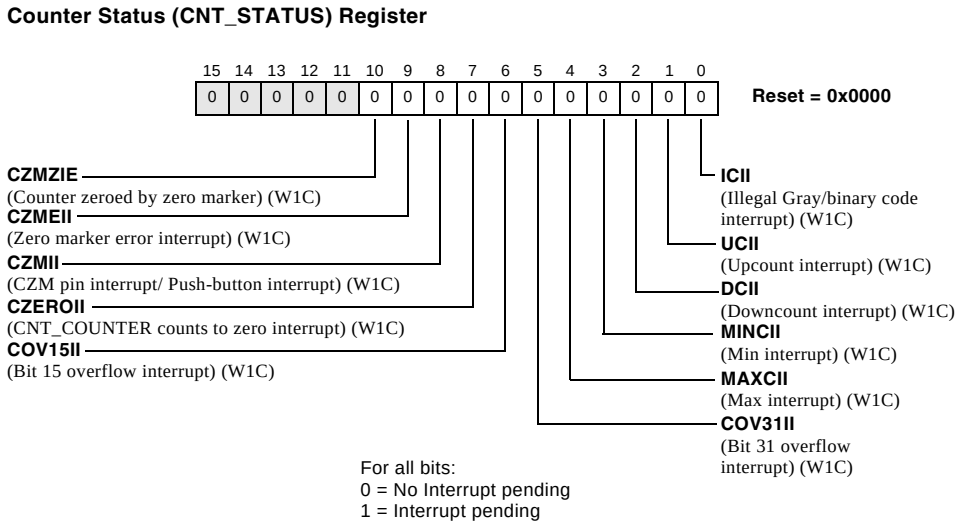


Figure 13-7. Counter Status Register

Counter Command Register (CNT_COMMAND)

The CNT_COMMAND register (shown in Figure 13-8 on page 13-24) configures the GP counter, enabling operations such as zeroing a counter register and copying or swapping boundary registers. These actions are taken by writing a “one” to the appropriate bit.

Read operations from this register will not return meaningful values, with the exception of the W1ZONCE bit, where a “1” indicates that the bit has been set by software before, but no zero marker event has been detected on the CZM pin yet. Refer to the “Zero Marker (Push Button) Operation” on page 13-9 for more details.

The CNT_COUNTER, CNT_MIN and CNT_MAX registers can be initialized to zero by writing a “one” to the W1LCNT_ZERO, W1LMIN_ZERO and W1LMAX_ZERO

fields. In addition to clearing registers, CNT_COMMAND allows the boundary registers to be modified in a number of ways. The current counter value in CNT_COUNT can be captured and loaded into either of the two boundary registers CNT_MAX and CNT_MIN to create new boundary limits. This is performed by setting the W1LMAX_CNT and W1LMIN_CNT bits. Alternatively, the counter can be loaded from CNT_MAX or CNT_MIN via the W1LCNT_MAX and W1LCNT_MIN bits. It is also possible to transfer the current CNT_MAX value into CNT_MIN (or vice versa) through the W1LMIN_MAX and W1LMAX_MIN bits. The final supported operation is the ability to only have the zero marker clear the CNT_COUNT register once, as described in "Zero Marker (Push Button) Operation" on page 13-9.

It is possible for multiple actions to be performed simultaneously by setting multiple bits in the CNT_COMMAND register. However, there are restrictions. The bits associated with each command have been grouped together such that all bits that involve a write to the CNT_COUNTER register are located within bits 3:0 of the CNT_COMMAND register. All commands that involve a write to the CNT_MIN register are located within bits 7:4 of the CNT_COMMAND register, and all commands that involve a write to the CNT_MAX register are located within bits 11:8 of the CNT_COMMAND register.

A maximum of three commands can be issued at any one time, excluding the W1ZMONCE command. Note that (W1LCNT_MIN, W1LCNT_MAX and W1LCNT_ZERO) have to be used exclusively. Never

set more than one of them at the same time. The same rule applies for (W1LMAX_MIN, W1LMAX_CNT and W1LMAX_ZERO) and for (W1LMIN_MAX, W1LMIN_CNT, and W1LMIN_ZERO).

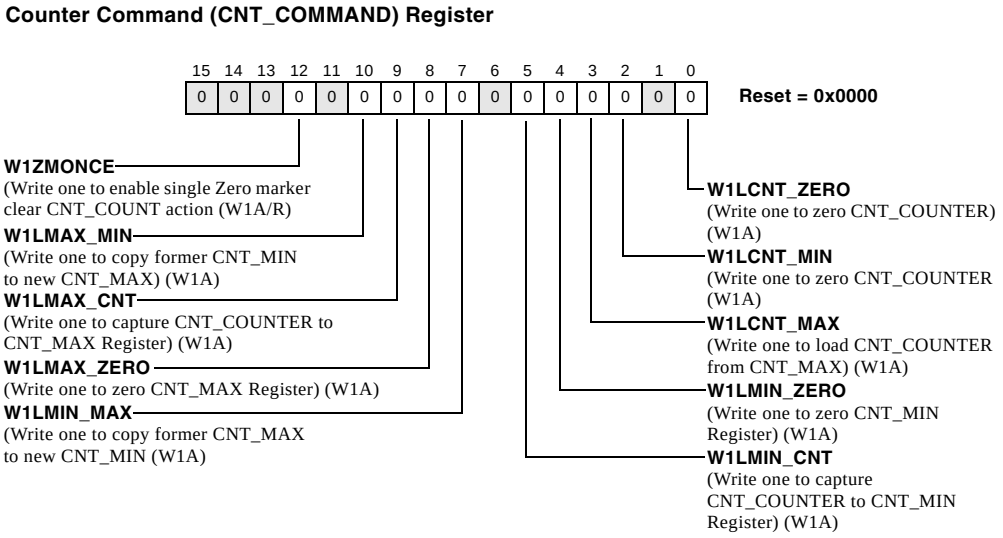

Counter Command (CNT_COMMAND) Register
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
W1ZMONCE
(Write one to enable single Zero marker clear CNT_COUNT action (W1A/R)
W1LMAX_MIN
(Write one to copy former CNT_MIN to new CNT_MAX) (W1A)
W1LMAX_CNT
(Write one to capture CNT_COUNTER to CNT_MAX Register) (W1A)
W1LMAX_ZERO
(Write one to zero CNT_MAX Register) (W1A)
W1LMIN_MAX
(Write one to copy former CNT_MAX to new CNT_MIN (W1A)
W1LCNT_ZERO
(Write one to zero CNT_COUNTER) (W1A)
W1LCNT_MIN
(Write one to zero CNT_COUNTER (W1A)
W1LCNT_MAX
(Write one to load CNT_COUNTER from CNT_MAX) (W1A)
W1LMIN_ZERO
(Write one to zero CNT_MIN Register) (W1A)
W1LMIN_CNT
(Write one to capture CNT_COUNTER to CNT_MIN Register) (W1A)

Figure 13-8. Counter Command Register

Counter Debounce Register (CNT_DEBOUNCE)

This register is used to select the noise filtering characteristic of the three input pins (see "Input Noise Filtering (Debouncing)" on page 13-8). Bits

[4:0] determine the filter time. The register can be accessed at any time with 16-bit read and write operations.

$$t_{filter} = 128 \times (2^{DPRESCALE} \div SCLK)$$

Counter Debounce (CNT_DEBOUNCE) Register

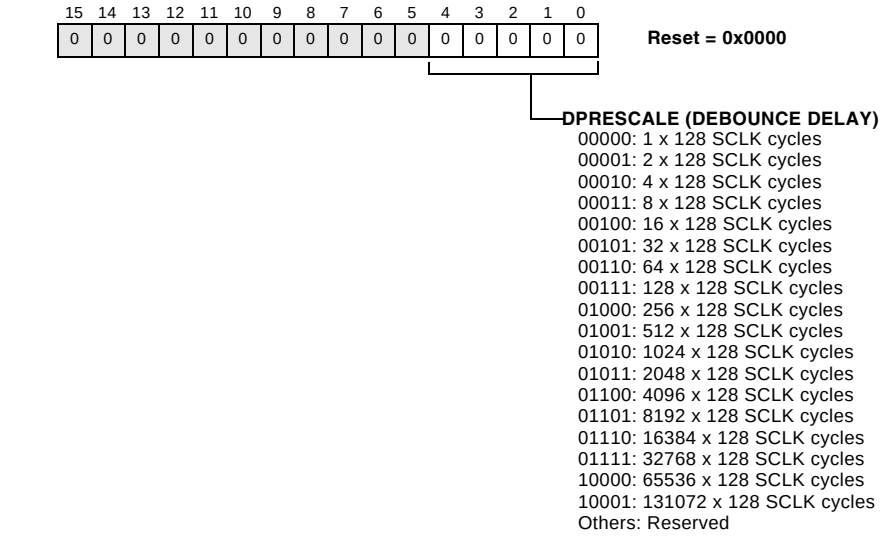


Figure 13-9. Counter Debounce Register

Counter Count Value Register (CNT_COUNTER)

This register holds the 32-bit, twos-complement, count value. It can be read and written at any time. Hardware ensures that reads and write are atomic, by providing respective shadow registers. This register can be

accessed with either 32-bit or 16-bit operations. This allows use of the GP counter as a 16-bit counter if sufficient for the application.

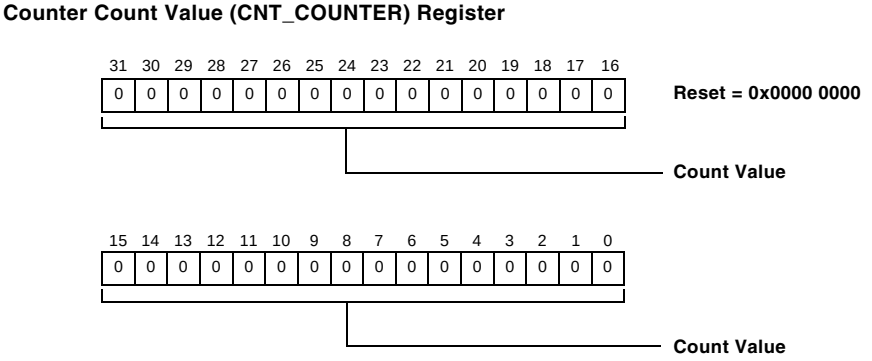


Figure 13-10. Counter Count Value Register

Counter Boundary Registers (CNT_MIN and CNT_MAX)

These registers hold the 32-bit, twos-complement, lower and upper boundary values. They can be read and written at any time. Hardware ensures that reads and write are atomic, by providing respective shadow registers. This register can be accessed with either 32-bit or 16-bit operations. This allows for using the GP counter as a 16-bit counter if sufficient for the application.

Counter Maximal Count (CNT_MAX) Register

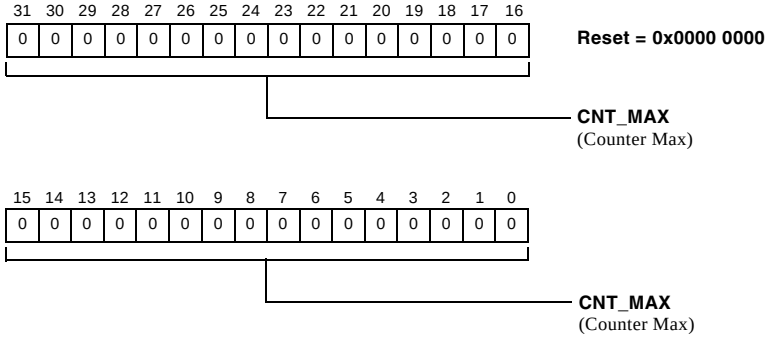


Figure 13-11. Counter Maximal Count Register

Counter Minimal Count (CNT_MIN) Register

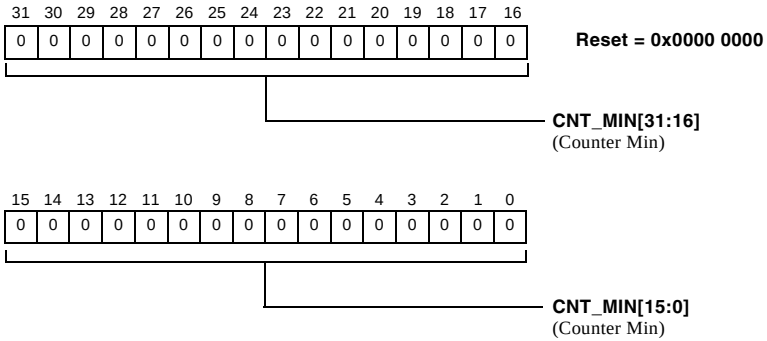


Figure 13-12. Counter Minimal Count Register

Programming Examples

Listing 13-1 illustrates how to initialize the GP counter for various modes. The required interrupts are first unmasked. The GP counter is then con-

figured for the required mode of operation. Note that at this point we do not yet enable the counter. Finally, some GP counter MMRs are cleared, as well as any interrupts that may be pending in the CNT_STATUS register.

Listing 13-1. Initializing the GP Counter

```
/* Setup Counter Interrupts */
P5.H = hi(CNT_IMASK);
P5.L = lo(CNT_IMASK);
R5 = nCZMZIE   /* Counter zeroed by zero marker interrupt */
    | CZMEIE   /* Zero marker error interrupt */
    | CZMIE    /* CZM pin interrupt (push-button) */
    | CZEROIE /* Counts to zero interrupt */
    | nCOV15IE /* Counter bit 15 overflow interrupt */
    | nCOV31IE /* Counter bit 31 overflow interrupt */
    | MAXCIE   /* Max count interrupt */
    | MINCIE   /* Min count interrupt */
    | DCIE     /* Downcount interrupt */
    | UCIE     /* Upcount interrupt */
    | ICIE (z);    /* Illegal gray/binary code interrupt */
w[P5] = R5;

/* Configure the GP Counter mode of operation */
P5.H = hi(CNT_CONFIG);
P5.L = lo(CNT_CONFIG);
R5 = nINPDIS           /* Enable CUD and CDG inputs */
    | BNDMODE_COMP     /* Boundary compare mode */
    | nZMZC            /* Disable Zero Counter Enable */
    | CNTMODE_QUADENC /* Quadrature Encoder Mode */
    | CZMINV           /* Polarity of CZM pin */
    | nCUDINV          /* Polarity of CUD pin */
    | nCDGINV          /* Polarity of CDG Pin */
    | nDEBE            /* Disable the debounce */
    | nCNTE (z);       /* Disable the counter */
```

```
w[P5] = R5;

/* Zero the CNT_COUNT, CNT_MIN and CNT_MAX registers
This is optional as after reset they are default to zero */
P5.H = hi(CNT_COMMAND);
P5.L = lo(CNT_COMMAND);
R5 = W1LCNT_ZERO | W1LMIN_ZERO | W1LMAX_ZERO (z);
w[P5] = R5;

/* Clear any identified interrupts */
P5.H = hi(CNT_STATUS);
P5.L = lo(CNT_STATUS);
R5.L = ICII     /* Illegal Gray/Binary Code Interrupt Identifier
*/
       | UCII   /* Up count Interrupt Identifier */
       | DCII   /* Down count Interrupt Identifier */
       | MINCII /* Min Count Interrupt Identifier */
       | MAXCII /* Max Count Interrupt Identifier */
       | COV31II /* Bit 31 Overflow Interrupt Identifier */
       | COV15II /* Bit 15 Overflow Interrupt Identifier */
       | CZEROII /* Count to Zero Interrupt Identifier */
       | CZMII   /* CZM Pin Interrupt Identifier */
       | CZMEII /* CZM Error Interrupt Identifier */
       | CZMZII; /* CZM Zeroes Counter Interrupt Identifier */
w[P5] = R5;
```

Listing 13-2 illustrates how to set up the peripheral and core interrupts for the GP counter. This example assumes the counter interrupts are generated on IRQ27, which is assumed to be mapped to the IVG11 interrupt. Finally, the system and peripheral interrupts are unmasked, and then the GP counter is enabled. This example can be easily tailored to processors with different SIC register mappings.

Listing 13-2. Setting Up the Interrupts for the GP Counter

```
/* Assign the CNT interrupt to IVG11 */
P5.H = hi(SIC_IAR3);
P5.L = lo(SIC_IAR3);
R6.H = hi(0xFFFF4FFF);
R6.L = lo(0xFFFF4FFF);
R7.H = hi(0x00000000);
R7.L = lo(0x00000000);
R5 = [P5];
R5 = R5 & R6; /* zero the counter interrupt field */
R5 = R5 | R7; /* set Counter interrupt to required priority */
[P5] = R5;

/* Set up the interrupt vector for the counter */
R5.H = hi(_IVG11_handler);
R5.L = lo(_IVG11_handler);
P5.H = hi(EVT11);
P5.L = lo(EVT11);
[P5] = R5;

/* Unmask IVG11 interrupt in the IMASK register */
P5.H = hi(IMASK);
P5.L = lo(IMASK);
R5 = [P5];
bitset(R5, bitpos(EVT_IVG11));
[P5] = R5;

/* Unmask interrupt 27 generated by the counter */
P5.H = hi(SIC_IMASK0);
P5.L = lo(SIC_IMASK0);
R5 = [P5];
bitset(R5, bitpos(IRQ_CNT));
[P5] = R5;
```

```
/* Enable the counter */
P5.H = hi(CNT_CONFIG);
P5.L = lo(CNT_CONFIG);
R5 = w[P5](z);
bitset(R5, bitpos(CNTE));
w[P5] = R5.L;
```

Using the same assumptions from the previous example, Listing 13-3 illustrates a sample interrupt handler that is responsible for servicing the GP counter interrupts. On entry to the handler, the SIC_ISR0 register is interrogated to determine if the counter is waiting for an interrupt to be serviced. If so, the handler responsible for processing all counter interrupts is called.

Listing 13-3. Sample Interrupt Handler for GP Counter Interrupts

```
_IVG11_handler:
    /* Stack management */
    [--SP] = RETS;
    [--SP] = ASTAT;
    [--SP] = (R7:0, P5:0);

    /* Was it a counter interrupt? */
    P5.H = hi(SIC_ISR0);
    P5.L = lo(SIC_ISR0);
    R5 = [P5];
    CC = bittst(R5, bitpos(IRQ_CNT));
    IF !CC JUMP _IVG11_handler.completed;
    CALL _IVG11_handler.counter;

    _IVG11_handler.completed:

    SSYNC;
```

```
    /* Restore from stack */
    (R7:0, P5:0) = [SP++];
    ASTAT = [SP++];
    RETS = [SP++];
    RTI;    /* Exit the interrupt service routine */
_IVG11_handler.end:

_IVG11_handler.counter:
    /* Stack management */
    [--SP] = RETS;
    [--SP] = (R7:0, P5:0);

    /* Determine what counter interrupts we wish to service */
    R5 = w[P5](z);
    P5.H = hi(CNT_IMASK);
    P5.L = lo(CNT_IMASK);
    R5 = w[P5](z);

    P5.H = hi(CNT_STATUS);
    P5.L = lo(CNT_STATUS);
    R6 = w[P5](z);
    R5 = R5 & R6;

/* Interrupt handlers for all GP counter interrupts */
_IVG11_handler.counter.illegal_code:
    CC = bittst(R5, bitpos(ICII));
    IF !CC JUMP _IVG11_handler.counter.up_count;

    /* Clear the serviced request */
    R6 = ICII (z);
    w[P5] = R6;

    /* insert illegal code handler here */
```

```
_IVG11_handler.counter.illegal_code.end:


_IVG11_handler.counter.up_count:
    CC = bittst(R5, bitpos(UCII));
    IF !CC JUMP _IVG11_handler.counter.down_count;

    /* Clear the serviced request */
    R6 = UCII (z);
    w[P5] = R6;

    /* insert up count handler here */

_IVG11_handler.counter.up_count.end:

_IVG11_handler.counter.down_count:
    CC = bittst(R5, bitpos(DCII));
    IF !CC JUMP _IVG11_handler.counter.min_count;

    /* Clear the serviced request */
    R6 = DCII (z);
    w[P5] = R6;

    /* insert down count handler here */

_IVG11_handler.counter.down_count.end:

_IVG11_handler.counter.min_count:
    CC = bittst(R5, bitpos(MINCII));
    IF !CC JUMP _IVG11_handler.counter.max_count;

    /* Clear the serviced request */
    R6 = MINCII (z);
    w[P5] = R6;
```

```
    /* insert min count handler here */

_IVG11_handler.counter.min_count.end:


_IVG11_handler.counter.max_count:
    CC = bittst(R5, bitpos(MAXCII));
    IF !CC JUMP _IVG11_handler.counter.b31_overflow;

    /* Clear the serviced request */
    R6 = MAXCII (z);
    w[P5] = R6;

    /* insert max count handler here */

_IVG11_handler.counter.max_count.end:

_IVG11_handler.counter.b31_overflow:
    CC = bittst(R5, bitpos(COV31II));
    IF !CC JUMP _IVG11_handler.counter.b15_overflow;

    /* Clear the serviced request */
    R6 = COV31II (z);
    w[P5] = R6;

    /* insert bit 31 overflow handler here */

_IVG11_handler.counter.b31_overflow.end:

_IVG11_handler.counter.b15_overflow:
    CC = bittst(R5, bitpos(COV15II));
    IF !CC JUMP _IVG11_handler.counter.count_to_zero;
```

```
    /* Clear the serviced request */
    R6 = COV15II (z);
    w[P5] = R6;

    /* insert bit 15 overflow handler here */

_IVG11_handler.counter.b15_overflow.end:

_IVG11_handler.counter.count_to_zero:
    CC = bittst(R5, bitpos(CZEROII));
    IF !CC JUMP _IVG11_handler.counter.czm;

    /* Clear the serviced request */
    R6 = CZEROII (z);
    w[P5] = R6;

    /* insert count to zero handler here */

_IVG11_handler.counter.count_to_zero.end:

_IVG11_handler.counter.czm:
    CC = bittst(R5, bitpos(CZMII));
    IF !CC JUMP _IVG11_handler.counter.czm_error;

    /* Clear the serviced request */
    R6 = CZMII (z);
    w[P5] = R6;

    /* insert czm handler here */

_IVG11_handler.counter.czm.end:

_IVG11_handler.counter.czm_error:
    CC = bittst(R5, bitpos(CZMEII));
```

```
    IF !CC JUMP _IVG11_handler.counter.czm_zeroes_counter;

    /* Clear the serviced request */
    R6 = CZMEII (z);
    w[P5] = R6;

    /* insert czm error handler here */

_IVG11_handler.counter.czm_error.end:


_IVG11_handler.counter.czm_zeroes_counter:
    CC = bittst(R5, bitpos(CZMZII));
    IF !CC JUMP _IVG11_handler.counter.all_serviced;

    /* Clear the serviced request */
    R6 = CZMZII (z);
    w[P5] = R6;

    /* insert czm zeroes counter handler here */

_IVG11_handler.counter.czm_zeroes_counter.end:

_IVG11_handler.counter.all_serviced:

    /* Restore from stack */
    (R7:0, P5:0) = [SP++];
    RETS = [SP++];
    RTS;
_IVG11_handler.counter.end:
```

Listing 13-4 shows how to set up timer 7 (as an example) to capture the period of counter events. Refer to the "Internal Interfaces" section of Chapter 9, “General-Purpose Ports” for information regarding which GP timer(s) are associated with which GP counter module(s) for your device.

The timer is configured for WDTH_CAP mode, and the period between the last two successive counter events is read from within the up count interrupt handler that was provided in Listing 13-3 on page 13-31.

Listing 13-4. Setting Up Timer 7 for Counter Event Period Capture

```
/* configure the timer for WDTH_CAP mode */
P5.H = hi(TIMER7_CONFIG);
P5.l = lo(TIMER7_CONFIG);
R5 = PULSE_HI | PERIOD_CNT | TIN_SEL | WDTH_CAP (z);
w[P5] = R5.l;

/* Enable Timer 7
P5.H = hi(TIMER_ENABLE0);
P5.L = lo(TIMER_ENABLE0);
R5 = TIMEN7 (z);
w[P5] = R5.L;

...

_IVG11_handler.counter.up_count:
    CC = bittst(R5, bitpos(UCII));
    IF !CC JUMP _IVG11_handler.counter.down_count;

    /* Clear the serviced request */
    R6 = UCII (z);
    w[P5] = R6;

    /* insert up count handler here */

    /* Read the period between the last two successive events */
    P5.H = hi(TIMER7_PERIOD);
    P5.L = lo(TIMER7_PERIOD);
    R5 = [P5];
```

```
    P5.H = hi(_event_period);
    P5.L = lo(_event_period);
    [P5] = R5;
_IVG11_handler.counter.up_count.end:
```

Unique Behavior for the ADSP-BF52x Processor

None

14 REAL-TIME CLOCK

This chapter describes the real-time clock (RTC). Following an overview and list of key features is a description of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF52x

For RTC interrupt vector assignments, refer to Table 5-3 on page 5-18 in Chapter 5, "System Interrupts".

For a list of MMR addresses for the RTC, refer to Appendix A, "System MMR Assignments".

RTC behavior for the ADSP-BF52x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF52x Processor" on page 14-27

Overview

The RTC provides a set of digital watch features to the processor, including time of day, alarm, and stopwatch countdown. It is typically used to implement either a real-time watch or a life counter, which counts the elapsed time since the last system reset.

The RTC watch features are clocked by a 32.768 kHz crystal external to the processor. The RTC uses dedicated power supply pins and is indepen-

dent of any reset, which enables it to maintain functionality even when the rest of the processor is powered down.

The RTC input clock is divided down to a 1 Hz signal by a prescaler, which can be bypassed. When bypassed, the RTC is clocked at the 32.768 kHz crystal rate. In normal operation, the prescaler is enabled.

The primary function of the RTC is to maintain an accurate day count and time of day. The RTC accomplishes this by means of four counters:

- 60 second counter
- 60 minute counter
- 24 hour counter
- 32768 day counter

The RTC increments the 60 second counter once per second and increments the other three counters when appropriate. The 32768 day counter is incremented each day at midnight (0 hours, 0 minutes, 0 seconds). Interrupts can be issued periodically, either every second, every minute, every hour, or every day. Each of these interrupts can be independently controlled.

The RTC provides two alarm features, programmed with the `RTC_ALARM` register. The first is a time of day alarm (hour, minute, and second). When the alarm interrupt is enabled, the RTC generates an interrupt each day at the time specified. The second alarm feature allows the application to specify a day as well as a time. When the day alarm interrupt is enabled, the RTC generates an interrupt on the day and time specified. The alarm interrupt and day alarm interrupt can be enabled or disabled independently.

The RTC provides a stopwatch function that acts as a countdown timer. The application can program a second count into the RTC stopwatch count register (`RTC_SWCNT`). When the stopwatch interrupt is enabled and

the specified number of seconds have elapsed, the RTC generates an interrupt.

Interface Overview

The RTC external interface consists of two clock pins, which together with the external components form the reference clock circuit for the RTC. The RTC interfaces internally to the processor system through the peripheral access bus (PAB), and through the interrupt interface to the system interrupt controller (SIC).

The RTC has dedicated power supply pins that power the clock functions at all times, including when the core power supply is turned off. Figure 14-1 provides a block diagram of the RTC.

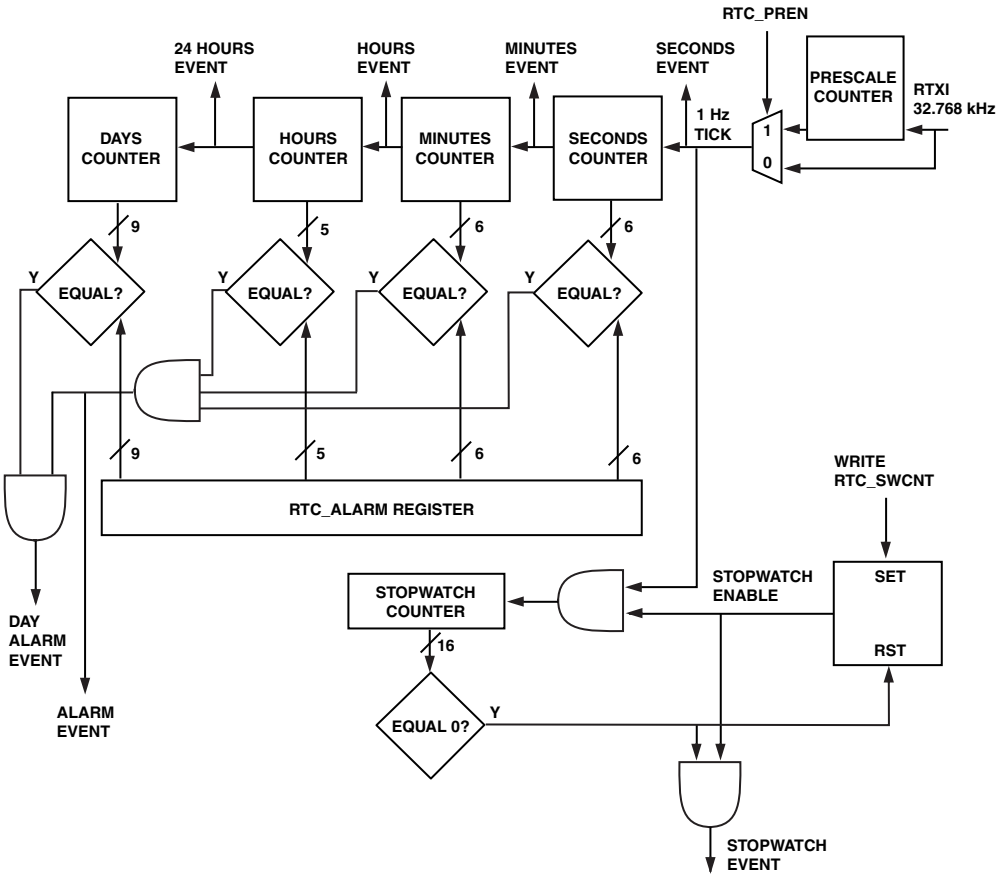


Figure 14-1. RTC Block Diagram

Description of Operation

The following sections describe the operation of the RTC.

RTC Clock Requirements

The RTC timer is clocked by a 32.768 kHz crystal external to the processor. The RTC system memory mapped registers (MMRs) are clocked by this crystal. When the prescaler is disabled, the RTC MMRs are clocked at the 32.768 kHz crystal frequency. When the prescaler is enabled, the RTC MMRs are clocked at the 1 Hz rate.

There is no way to disable the RTC counters using software. If a given system does not require the RTC functionality, then it may be disabled with hardware tie offs. Tie the RTXI and RTCGND pins to EGND, tie the RTCVDD pin to EVDD, and leave the RTXO pin unconnected. Additionally, writing RTC_PREN to "0" saves a small amount of power.

Prescaler Enable

The single active bit of the RTC prescaler enable register (RTC_PREN) is written using a synchronization path. Clearing of the bit is synchronized to the 32.768 kHz clock. This faster synchronization allows the module to be put into high speed mode (bypassing the prescaler) without waiting the full one second for the write to complete that would be necessary if the module were already running with the prescaler enabled. When this bit is cleared, the prescaler is disabled, and the RTC runs at the 32.768 kHz crystal frequency.

When setting the RTC_PREN bit, the first positive edge of the 1 Hz clock occurs 1 to 2 cycles of the 32.768 kHz clock after the prescaler is enabled. The write complete status/interrupt works as usual when enabling or disabling the prescale counter. The new RTC clock rate is in effect before the write complete status is set. In order for the RTC to operate at the proper rate, software must set the prescaler enable bit after initial powerup. When this bit is set, the prescaler is enabled, and the RTC runs at a frequency of 1 Hz.

Write RTC_PREN and then wait for the write complete event before programming the other registers. It is safe to write RTC_PREN to “1” every time the processor boots. The first time sets the bit, and subsequent writes have no effect, as no state is changed.

Do not disable the prescaler by clearing the bit in the RTC_PREN register without making sure that there are no writes to RTC MMRs in progress. Do not switch between fast and slow mode during normal operation by setting and clearing this bit, as this disrupts the accurate tracking of real time by the counters. To avoid these potential errors, initialize the RTC during startup using RTC_PREN and do not dynamically alter the state of the prescaler during normal operation.

Running without the prescaler enabled is provided primarily as a test mode. All functionality works, just 32,768 times as fast. Typical software should never program RTC_PREN to “0”. The only reason to do so is to synchronize the 1 Hz tick to a more precise external event, as the 1 Hz tick predictably occurs a few RTXI cycles after a 0-to-1 transition of RTC_PREN.

Use the following sequence to achieve synchronization to within 100 ms.

1. Write RTC_PREN to “0”.
2. Wait for the write to complete.
3. Wait for the external event.
4. Write RTC_PREN to “1”.
5. Wait for the write to complete.
6. Reprogram the time into RTC_STAT.

RTC Programming Model

The RTC programming model consists of a set of system MMRs. Software can configure the RTC and can determine the status of the RTC through reads and writes to these registers. The RTC interrupt control register (RTC_ICTL) and the RTC interrupt status register (RTC_ISTAT) provide RTC interrupt management capability.

Note that software cannot disable the RTC counting function. However, all RTC interrupts can be disabled, or masked. At reset, all interrupts are disabled. The RTC state can be read via the system MMR status registers at any time.

The primary RTC functionality, shown in Figure 14-1 on page 14-4, consists of registers and counters that are powered by an independent RTC supply (V_{DDRTC}). This logic is never reset; it comes up in an indeterminate state when V_{DDRTC} is first powered on.

The RTC also contains logic powered by the same internal V_{dd} as the processor core and other peripherals. This logic contains some control functionality, holding registers for PAB write data, and prefetched PAB read data shadow registers for each of the five V_{DDRTC}-powered registers. This logic is reset by the same system reset and clocked by the same SCLK as the other peripherals.

Figure 14-2 shows the connections between the V_{DDRTC}-powered RTC MMRs and their corresponding V_{DDINT}-powered write holding registers and read shadow registers. In the figure, “REG” means each of the RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, and RTC_PREN registers. The RTC_ISTAT register connects only to the PAB.

The rising edge of the 1 Hz RTC clock is the “1 Hz tick”. Software can synchronize to the 1 Hz tick by waiting for the seconds event flag to set or by waiting for the seconds interrupt (if enabled).

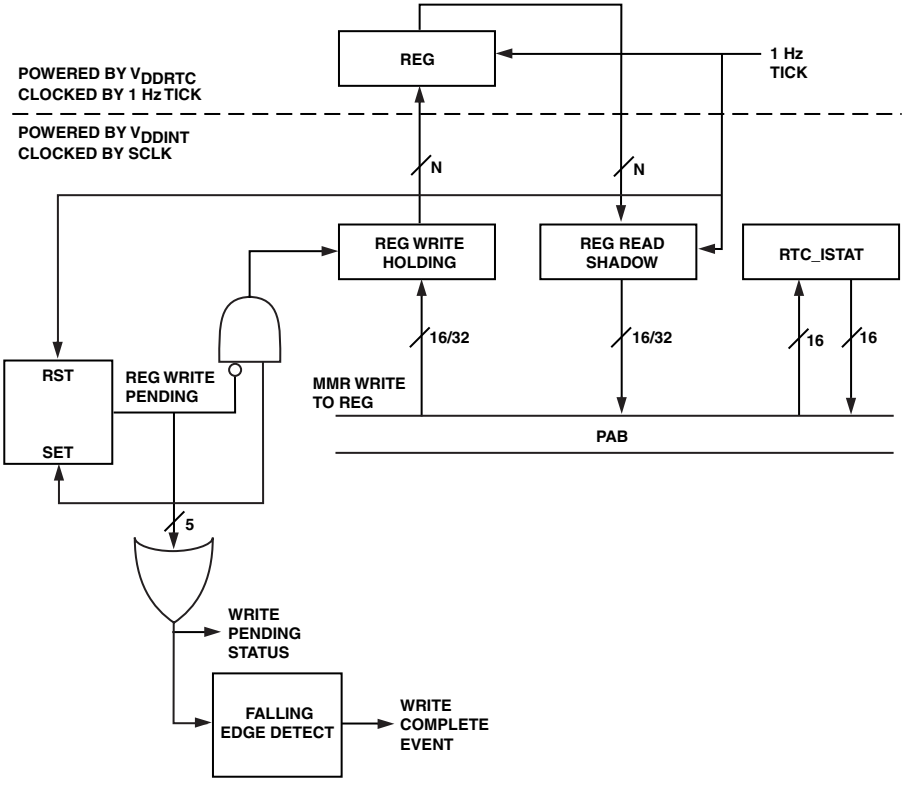


Figure 14-2. RTC Register Architecture

Register Writes

Writes to all RTC MMRs except RTC_ISTAT are saved in write holding registers and then are synchronized to the RTC 1 Hz clock. The write pending status bit in RTC_ISTAT indicates the progress of the write. The write pending status bit is set when a write is initiated and is cleared when all writes are complete. The falling edge of the write pending status bit causes the write complete flag in RTC_ISTAT to be set. This flag can be configured in RTC_ICTL to cause an interrupt. Software does not have to wait for writes to one RTC MMR to complete before writing to another RTC

MMR. The write pending status bit is set if any writes are in progress, and the write complete flag is set only when all writes are complete.

Any writes in progress when peripherals are reset are aborted. Do not stop SCLK (for example, by entering deep sleep mode) or remove internal V_{dd} power until all RTC writes have completed.

Do not attempt another write to the same register without waiting for the previous write to complete. Subsequent writes to the same register are ignored if the previous write is not complete.

Reading a register that has been written before the write complete flag in RTC_ISTAT is set will return the old value. Always check the write pending status bit in RTC_ISTAT before attempting a read or write.

Write Latency

Writes to the RTC MMRs are synchronized to the 1 Hz RTC clock. When setting the time of day, do not factor in the delay when writing to the RTC MMRs. The most accurate method of setting the RTC is to monitor the seconds (1 Hz) event flag or to program an interrupt for this event and then write the current time to RTC_STAT in the interrupt service routine (ISR). The new value is inserted ahead of the incrementer. Hardware adds one second to the written value (with appropriate carries into minutes, hours and days) and loads the incremented value at the next 1 Hz tick, when it represents the then-current time.

Writes posted at any time are properly synchronized to the 1 Hz clock. Writes complete at the rising edge of the 1 Hz clock. A write posted just before the 1 Hz tick may not be completed until the 1 Hz tick one second later. Any write posted in the first 990 ms after a 1 Hz tick completes on the next 1 Hz tick, but the simplest, most predictable and recommended technique is to only post writes to RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, or RTC_PREN immediately after a seconds interrupt or event. All five registers may be written in the same second.

W1C bits in the RTC_ISTAT register take effect immediately.

Register Reads

There is no latency when reading RTC MMRs, as the values come from the read shadow registers. These shadow registers are updated and ready for reading by the time any RTC interrupts or event flags for that second are asserted. Once the internal V_{dd} logic completes its initialization sequence after SCLK starts, there is no point in time when it is unsafe to read the RTC MMRs for synchronization reasons. They always return coherent values, although the values may be indeterminate.

Deep Sleep

When the dynamic power management mode is set to deep sleep, all clocks in the system (except RTXI and the RTC 1 Hz tick) are stopped. In this state, the V_{DDRTC}-powered counters continue to increment. The internal V_{dd} shadow registers are not updated, but neither can they be read.

During deep sleep mode, all bits in RTC_ISTAT are cleared. Events that occur during deep sleep are not recorded in RTC_ISTAT. The internal V_{dd} RTC control logic generates a virtual 1 Hz tick within one RTXI period (30.52 µs) after SCLK restarts. This loads all shadow registers with up-to-date values and sets the seconds event flag. Other event flags may also be set. When the system wakes up from deep sleep, whether by an RTC event or a hardware reset, all of the RTC events that occurred during that second (and only that second) are reported in RTC_ISTAT.

When the system wakes up from deep sleep mode, software does not need to W1C the bits in RTC_ISTAT. All W1C bits are already cleared by hardware. The seconds event flag is set when the RTC internal V_{dd} logic has completed its restart sequence. Software should wait until the seconds event flag is set and then may begin reading or writing any RTC register.

Event Flags

The indeterminate values in the registers at power-up can cause event flags to set before the correct value is written into each of the registers. By catching the 1 Hz clock edge, the write to RTC_STAT can occur a full second before the write to RTC_ALARM. This would cause an extra second of delay between the validity of RTC_STAT and RTC_ALARM, if the value of the RTC_ALARM out of reset is the same as the value written to RTC_STAT. Therefore, wait for the writes to complete on these registers before using the flags and interrupts associated with their values.

The following is a list of flags along with the conditions under which they are valid:

- Seconds (1 Hz) event flag
  Always set on the positive edge of the 1 Hz clock and after shadow registers have updated after waking from deep sleep. This is valid as long as the RTC 1 Hz clock is running. Use this flag or interrupt to validate the other flags.
- Write complete and write pending status
  Always valid.
- Minutes event flag
  Valid only after the second field in RTC_STAT is valid. Use the write complete and write pending status flags or interrupts to validate the RTC_STAT value before using this flag value or enabling the interrupt.
- Hours event flag
  Valid only after the minute field in RTC_STAT is valid. Use the write complete and write pending status flags or interrupts to validate the RTC_STAT value before using this flag value or enabling the interrupt.

- 24 Hours event flag
  Valid only after the hour field in RTC_STAT is valid. Use the write complete and write pending status flags or interrupts to validate the RTC_STAT value before using this flag value or enabling the interrupt.
- Stopwatch event flag
  Valid only after the RTC_SWCNT register is valid. Use the write complete and write pending status flags or interrupts to validate the RTC_SWCNT value before using this flag value or enabling the interrupt.
- Alarm event and day alarm event flags
  Valid only after the RTC_STAT and RTC_ALARM registers are valid. Use the write complete and write pending status flags or interrupts to validate the RTC_STAT and RTC_ALARM values before using this flag value or enabling its interrupt.

Writes posted together at the beginning of the same second take effect together at the next 1 Hz tick. The following sequence is safe and does not result in any spurious interrupts from a previous state.

1. Wait for 1 Hz tick.
2. Write 1s to clear the RTC_ISTAT flags for alarm, day alarm, stopwatch, and/or per interval.
3. Write new values for RTC_STAT, RTC_ALARM, and/or RTC_SWCNT.
4. Write new value for RTC_ICTL with alarm, day alarm, stopwatch, and/or per interval interrupts enabled.
5. Wait for 1 Hz tick.
6. New values have now taken effect simultaneously.

Setting Time of Day

The RTC_STAT register is used to read or write the current time. Reads return a 32-bit value that always reflects the current state of the days, hours, minutes, and seconds counters. Reads and writes must be 32-bit transactions; attempted 16-bit transactions result in an MMR error. Reads always return a coherent 32-bit value. The hours, minutes, and seconds fields are usually set to match the real time of day. The day counter value is incremented every day at midnight to record how many days have elapsed since it was last modified. Its value does not correspond to a particular calendar day. The 15-bit day counter provides a range of 89 years, 260 or 261 days (depending on leap years) before it overflows.

After the 1 Hz tick, program RTC_STAT with the current time. At the next 1 Hz tick, RTC_STAT takes on the new, incremented value. For example:

1. Wait for 1 Hz tick.
2. Read RTC_STAT, get 10:45:30.
3. Write RTC_STAT to current time, 13:10:59.
4. Read RTC_STAT, still get old time 10:45:30.
5. Wait for 1 Hz tick.
6. Read RTC_STAT, get new current time, 13:11:00.

Using the Stopwatch

The RTC_SWCNT register contains the countdown value for the stopwatch. The stopwatch counts down seconds from the programmed value and generates an interrupt (if enabled) when the count reaches “0”. The counter stops counting at this point and does not resume counting until a new value is written to RTC_SWCNT. Once running, the counter may be overwritten with a new value. This allows the stopwatch to be used as a watchdog timer with a precision of one second.

The stopwatch can be programmed to any value between 0 and (2^{16} – 1) seconds, which is a range of 18 hours, 12 minutes, and 15 seconds.

Typically, software should wait for a 1 Hz tick, then write RTC_SWCNT. One second later, RTC_SWCNT changes to the new value and begins decrementing. Because the register write occupies nearly one second, the time from writing a value of N until the stopwatch interrupt is nearly N + 1 seconds. To produce an exact delay, software can compensate by writing N – 1 to get a delay of nearly N seconds. This implies that you cannot achieve a delay of 1 second with the stopwatch. Writing a value of "1" immediately after a 1 Hz tick results in a stopwatch interrupt nearly two seconds later. To wait one second, software should just wait for the next 1 Hz tick.

The RTC_SWCNT register is not reset. After initial powerup, it may be running. When the stopwatch is not used, writing it to "0" to force it to stop saves a small amount of power.

Interrupts

The RTC can provide interrupts at several programmable intervals:

- Per second, minute, hour, and day—based on increments to the respective counters in RTC_STAT
- On countdown from a programmable value—value in RTC_SWCNT transitions to "0" or is written with "0" by software (whether it was previously running or already stopped with a count of "0")
- Daily at a specific time—all fields of RTC_ALARM must match RTC_STAT except the day field
- On a specific day and time—all fields of RTC_ALARM register must match RTC_STAT

The RTC can be programmed to provide an interrupt at the completion of all pending writes to any of the 1 Hz registers (RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, and RTC_PREN). The eight RTC interrupt events can

be individually masked or enabled by the RTC_ICTL register. The seconds interrupt is generated on each 1 Hz clock tick, if enabled. The minutes interrupt is generated at the 1 Hz clock tick that advances the seconds counter from 59 to 0. The hour interrupt is generated at the 1 Hz clock tick that advances the minute counter from 59 to 0. The 24 hour interrupt occurs once per 24 hour period at the 1 Hz clock tick that advances the time to midnight (00:00:00). Any of these interrupts can generate a wakeup request to the processor, if enabled. All implemented bits are read/write.

This register is only partially cleared at reset, so some events may appear to be enabled initially. However, the RTC interrupt and the RTC wakeup to the PLL are handled specially and are masked (forced low) until after the first write to the RTC_ICTL register is complete. Therefore, all interrupts act as if they were disabled at system reset (as if all bits of RTC_ICTL were zero), even though some bits of RTC_ICTL may read as nonzero. If no RTC interrupts are needed immediately after reset, it is recommended to write RTC_ICTL to 0x0000 so that later read-modify-write accesses function as intended.

Interrupt status can be determined by reading the RTC_ISTAT register. All bits in RTC_ISTAT are sticky. Once set by the corresponding event, each bit remains set until cleared by a software write to this register. Event flags are always set; they are not masked by the interrupt enable bits in RTC_ICTL. Values are cleared by writing a “1” to the respective bit location, except for the write pending status bit, which is read-only. Writes of “0” to any bit of the register have no effect. This register is cleared at reset and during deep sleep.

The RTC interrupt is set whenever an event latched into the RTC_ISTAT register is enabled in the RTC_ICTL register. The pending RTC interrupt is cleared whenever all enabled and set bits in RTC_ISTAT are cleared, or when all bits in RTC_ICTL corresponding to pending events are cleared.

As shown in Figure 14-3, the RTC generates an interrupt request (IRQ) to the processor core for event handling and wakeup from a sleep state.

The RTC generates a separate signal for wakeup from a deep sleep or from an internal V_{dd} power-off state. The deep sleep wakeup signal is asserted at the 1 Hz tick when any RTC interval event enabled in RTC_ICTL occurs. The assertion of the deep sleep wakeup signal causes the processor core clock (CCLK) and the system clock (SCLK) to restart. Any enabled event that asserts the RTC deep sleep wakeup signal also causes the RTC IRQ to assert once SCLK restarts.

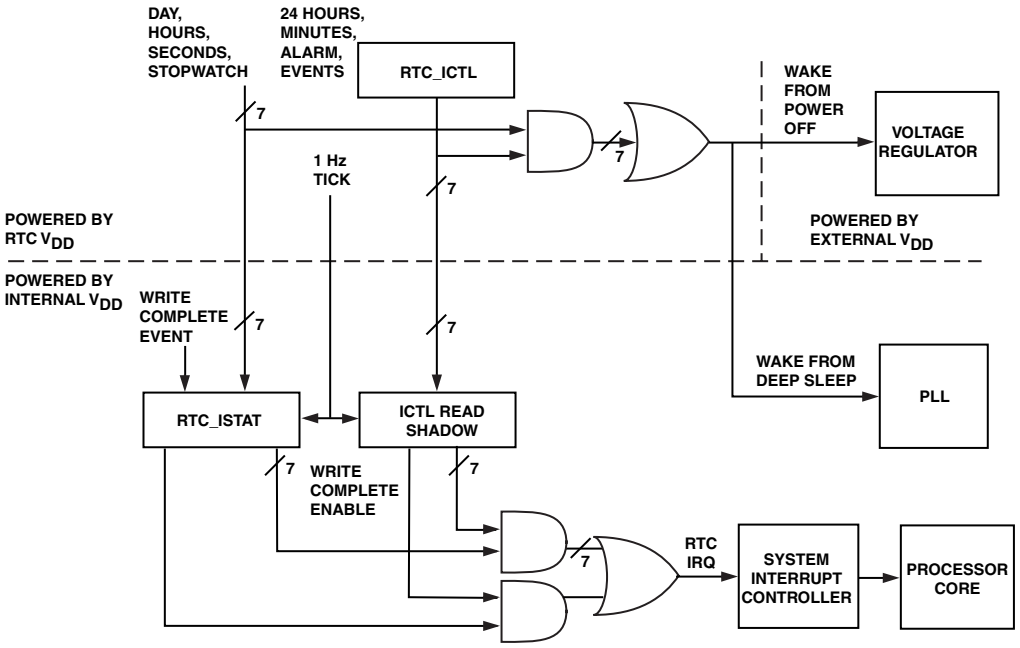


Figure 14-3. RTC Interrupt Structure

State Transitions Summary

Table 14-1 shows how each RTC MMR is affected by the system states. The phase locked loop (PLL) states are defined in the *Dynamic Power Management* chapter. “No power” means none of the processor power supply pins are connected to a source of energy. “Off” means the processor

core, peripherals, and memory are not powered (internal V_{dd} is off), while the RTC is still powered and running. External V_{dd} may still be powered. Registers described as “as written” are holding the last value software wrote to the register. If the register has not been written since V_{DDRTC} power was applied, then the state is indeterminate (for all bits of RTC_STAT, RTC_ALARM, and RTC_SWCNT, and for some bits of RTC_ISTAT, RTC_PREN, and RTC_ICTL).

Table 14-1. Effect of States on RTC MMRs

RTC V_{dd}	IV_{dd}	System State	RTC_ICTL	RTC_ISTAT	RTC_STAT RTC_SWCNT	RTC_ALARM RTC_PREN
Off	Off	No power	X	X	X	X
On	On	Reset	As written	0	Counting	As written
On	On	Full on	As written	Events	Counting	As written
On	On	Sleep	As written	Events	Counting	As written
On	On	Active	As written	Events	Counting	As written
On	On	Deep sleep	As written	0	Counting	As written
On	Off	Off	As written	X	Counting	As written

Table 14-2 summarizes software’s responsibilities with respect to the RTC at various system state transition events.

Table 14-2. RTC System State Transition Events

At This Event:	Execute This Sequence:
Power on from no power	Write RTC_PREN = 1. Wait for write complete. Write RTC_STAT to current time. Write RTC_ALARM, if needed. Write RTC_SWCNT. Write RTC_ISTAT to clear any pending RTC events. Write RTC_ICTL to enable any desired RTC interrupts or to disable all RTC interrupts.

Table 14-2. RTC System State Transition Events (Continued)

At This Event:	Execute This Sequence:
Full on after reset or Full on after power on from off	Wait for seconds event, or write RTC_PREN = 1 and wait for write complete. Write RTC_ISTAT to clear any pending RTC events. Write RTC_ICTL to enable any desired RTC interrupts or to disable all RTC interrupts. Read RTC MMRs as required.
Wake from deep sleep	Wait for seconds event flag to set. Write RTC_ISTAT to acknowledge RTC deep sleep wakeup. Read RTC MMRs as required. The PLL state is now active. Transition to full on as needed.
Wake from sleep	If wakeup came from RTC, seconds event flag will be set. In this case, write RTC_ISTAT to acknowledge RTC wakeup IRQ. Always, read RTC MMRs as required.
Before going to sleep	If wakeup by RTC is desired: Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable the desired RTC interrupt sources for wakeup. Wait for write complete. Enable RTC for wakeup in the system interrupt wakeup enable register (SIC_IWR).
Before going to deep sleep	Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable the desired RTC event sources for deep sleep wakeup. Wait for write complete.
Before going to off	Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable any desired RTC event sources for powerup wakeup. Wait for write complete. Set the wake bit in the voltage regulator control register (VR_CTL).

Register Definitions

The following sections contain the register definitions. Figure 14-4 through Figure 14-9 on page 14-22 illustrate the registers.

Table 14-3 shows the functions of the RTC registers.

Table 14-3. RTC Register Mapping

Register Name	Function	Notes
RTC_STAT	RTC status register	Holds time of day
RTC_ICTL	RTC interrupt control register	Bits 14:7 are reserved
RTC_ISTAT	RTC interrupt status register	Bits 13:7 are reserved
RTC_SWCNT	RTC stopwatch count register	Undefined at reset
RTC_ALARM	RTC alarm register	Undefined at reset
RTC_PREN	Prescaler enable register	Always set PREN = 1 for 1 Hz ticks

RTC Status (RTC_STAT) Register

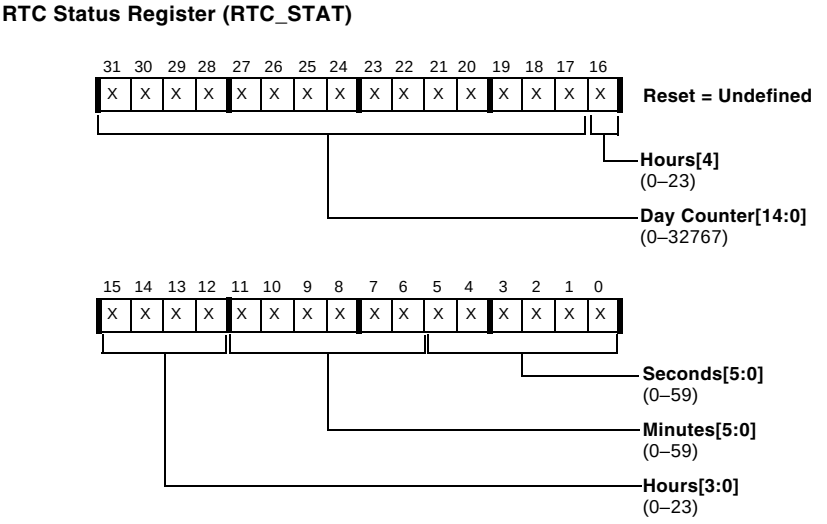

RTC Status Register (RTC_STAT)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Hours[4]
(0–23)
Day Counter[14:0]
(0–32767)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Seconds[5:0]
(0–59)
Minutes[5:0]
(0–59)
Hours[3:0]
(0–23)

Figure 14-4. RTC Status Register

RTC Interrupt Control (RTC_ICTL) Register

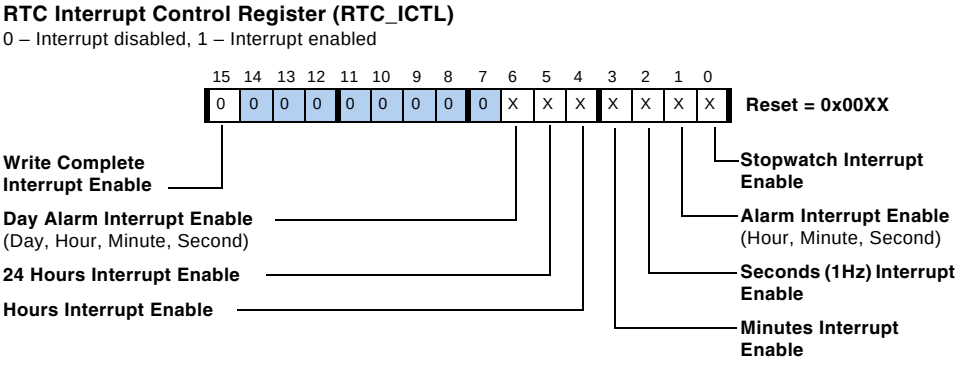

RTC Interrupt Control Register (RTC_ICTL)
0 – Interrupt disabled, 1 – Interrupt enabled
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 X X X X X X X
Reset = 0x00XX
Write Complete Interrupt Enable
Day Alarm Interrupt Enable
(Day, Hour, Minute, Second)
24 Hours Interrupt Enable
Hours Interrupt Enable
Stopwatch Interrupt Enable
Alarm Interrupt Enable
(Hour, Minute, Second)
Seconds (1Hz) Interrupt Enable
Minutes Interrupt Enable

Figure 14-5. RTC Interrupt Control Register

RTC Interrupt Status (RTC_ISTAT) Register

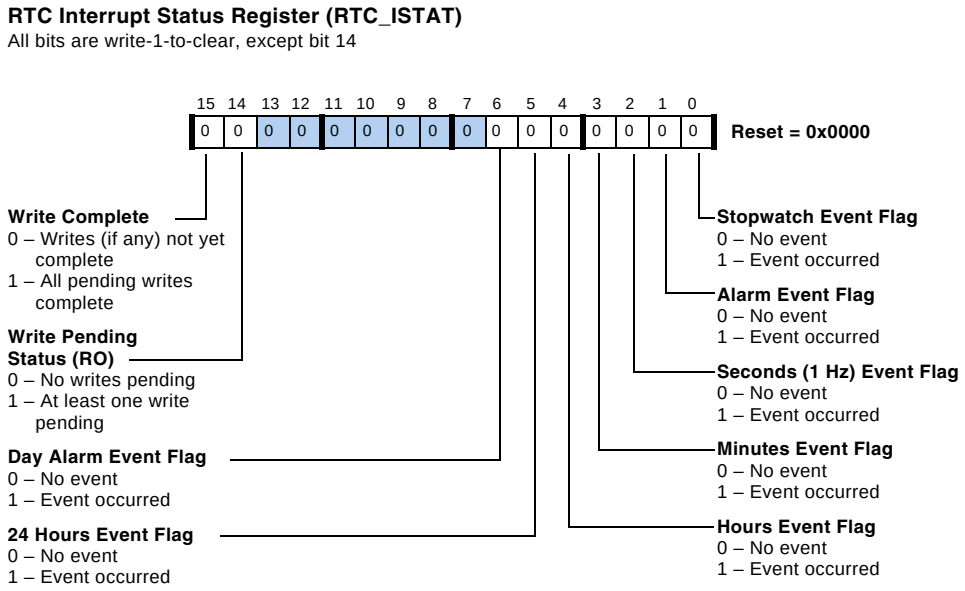

RTC Interrupt Status Register (RTC_ISTAT)
All bits are write-1-to-clear, except bit 14
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Write Complete
0 – Writes (if any) not yet complete
1 – All pending writes complete
Write Pending Status (RO)
0 – No writes pending
1 – At least one write pending
Day Alarm Event Flag
0 – No event
1 – Event occurred
24 Hours Event Flag
0 – No event
1 – Event occurred
Stopwatch Event Flag
0 – No event
1 – Event occurred
Alarm Event Flag
0 – No event
1 – Event occurred
Seconds (1 Hz) Event Flag
0 – No event
1 – Event occurred
Minutes Event Flag
0 – No event
1 – Event occurred
Hours Event Flag
0 – No event
1 – Event occurred

Figure 14-6. RTC Interrupt Status Register

RTC Stopwatch Count (RTC_SWCNT) Register

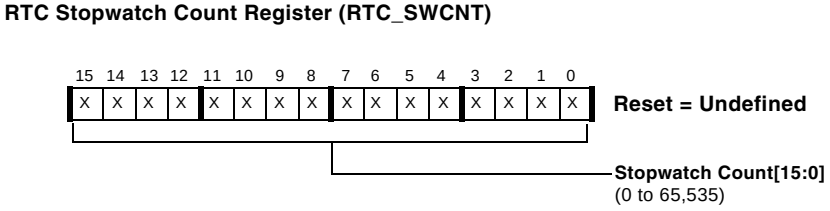

RTC Stopwatch Count Register (RTC_SWCNT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Reset = Undefined
Stopwatch Count[15:0]
(0 to 65,535)

Figure 14-7. RTC Stopwatch Count Register

RTC Alarm (RTC_ALARM) Register

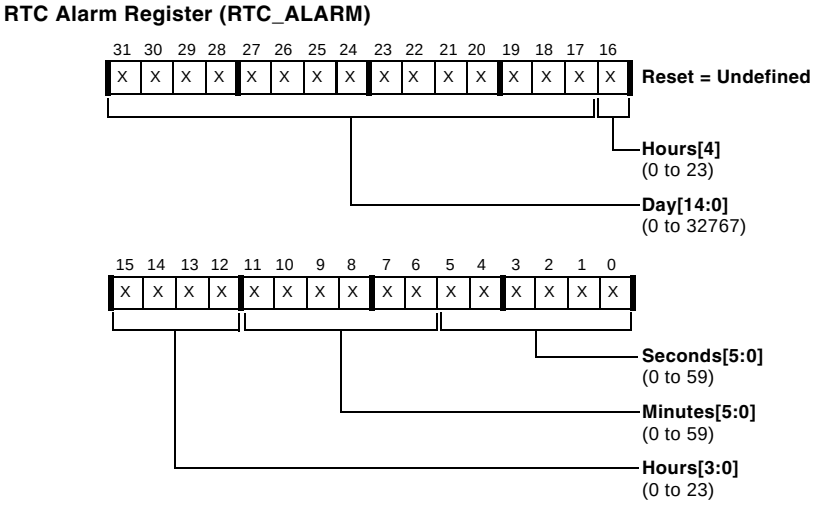

RTC Alarm Register (RTC_ALARM)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Hours[4]
(0 to 23)
Day[14:0]
(0 to 32767)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Seconds[5:0]
(0 to 59)
Minutes[5:0]
(0 to 59)
Hours[3:0]
(0 to 23)

Figure 14-8. RTC Alarm Register

RTC Prescaler Enable (RTC_PREN) Register

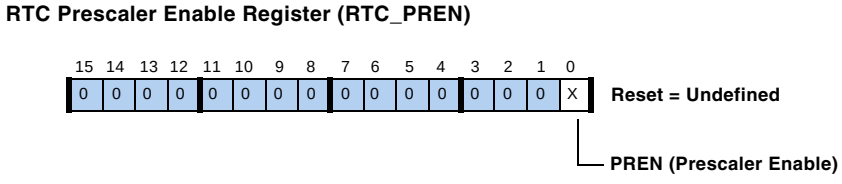

RTC Prescaler Enable Register (RTC_PREN)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 X
Reset = Undefined
PREN (Prescaler Enable)

Figure 14-9. RTC Prescaler Enable Register

Programming Examples

The following RTC code examples show how to enable the RTC prescaler, how to set up a stopwatch event to take the RTC out of deep sleep

mode, and how to use the RTC alarm to exit hibernate state. Each of these code examples assumes that the appropriate header file is included in the source code (for instance, `#include <defBF525.h>` for ADSP-BF525 projects).

Enable RTC Prescaler

Listing 14-1 properly enables the prescaler and clears any pending interrupts.

Listing 14-1. Enabling the RTC Prescaler

```
RTC_Initialization:
  P0.H = HI(RTC_PREN);
  P0.L = LO(RTC_PREN);
  R0=PREN(Z);  /* enable prescaler for 1 Hz ticks */
  W[P0] = R0.L;

  P0.L = LO(RTC_ISTAT);
  R0 = 0x807F(Z);
  W[P0] = R0.L;   /* clear any pending interrupts */

  R0 = WRITE_COMPLETE(Z);   /* mask for WRITE-COMPLETE bit */
  Poll_WC:   R1 = W[P0](Z);
             R1 = R1 & R0;  /* wait for Write Complete */
             CC = AZ;
             IF CC JUMP Poll_WC;
  RTS;
```

RTC Stopwatch For Exiting Deep Sleep Mode

Listing 14-2 sets up the RTC to utilize the stopwatch feature to come out of deep sleep mode. This code assumes that the `_RTC_Interrupt` label is properly registered as the ISR destination for the real-time clock event, the

RTC interrupt is enabled in both IMASK and SIC_IMASK, and that the RTC prescaler has already been enabled properly.

Listing 14-2. RTC Stopwatch Interrupt to Exit Deep Sleep

```
/* RTC Wake-Up Interrupt To Be Used With Deep Sleep Code */
_RTC_Interrupt:
  P0.H = HI(PLL_CTL);
  P0.L = LO(PLL_CTL);
  R0 = W[P0](Z);
  BITCLR (R0, BITPOS(BYPASS));
  W[P0] = R0;  /* If BYPASS Set, Must Clear It */

  IDLE;  /* Must go to IDLE for PLL changes to be effected */

  R0 = 0x807F(Z);
  P0.H = HI(RTC_ISTAT);
  P0.L = LO(RTC_ISTAT);
  W[P0] = R7;   /* clear pending RTC IRQs */

  R0 = WRITE_COMPLETE(Z);  /* mask for WRITE-COMPLETE bit */
  Poll_WC_IRQ:   R1 = W[P0](Z);
                 R1 = R1 & R0;  /* wait for Write Complete */
                 CC = AZ;
                 IF CC JUMP Poll_WC_IRQ;
  RTI;

Deep_Sleep_Code:
  P0.H = HI(RTC_SWCNT);
  P0.L = LO(RTC_SWCNT);
  R1 = 0x0010(Z);  /* set stop-watch to 16 seconds */
  W[P0] = R1.L;  /* will produce ~15 second delay */

  P0.L = LO(RTC_ICTL);
```

```
  R1 = STOPWATCH(Z);
  W[P0] = R1.L;  /* enable Stop-Watch interrupt */
  P0.L = LO(RTC_ISTAT);
  R1 = 0x807F(Z);
  W[P0] = R1.L;  /* clear any pending RTC interrupts */

  R0 = WRITE_COMPLETE(Z);  /* mask for WRITE-COMPLETE bit */
  Poll_WC1:   R1 = W[P0](Z);
              R1 = R1 & R0;  /* wait for Write Complete */
              CC = AZ;
              IF CC JUMP Poll_WC1;

/* RTC now running with correct stop-watch count and interrupts
*/
  P0.H = HI(PLL_CTL);
  P0.L = LO(PLL_CTL);
  R0 = W[P0](Z);
  BITSET (R0, BITPOS(PDWN));  /* set PDWN To Go To Deep Sleep */
  W[P0] = R0.L;  /* Issue Command for Deep Sleep */

  CLI R0;        /* Perform PLL Programming Sequence */
  IDLE;
  STI R0;        /* In Deep Sleep When Idle Exits */

  RTS;
```

RTC Alarm to Come Out of Hibernate State

Listing 14-3 sets up the RTC to utilize the alarm feature to come out of hibernate state. This code assumes that the prescaler has already been properly enabled.

Listing 14-3. Setting RTC Alarm to Exit Hibernate State

```
Hibernate_Code:
  P0.H = HI(RTC_ALARM);
  P0.L = LO(RTC_ALARM);
  R0 = 0x0010(Z);  /* set alarm to 16 seconds from now */
  W[P0] = R0.L;

  P0.L = LO(RTC_STAT);
  R0 = 0;  /* Clear RTC Status to Start Counting at 0 */
  W[P0] = R0.L;

  P0.L = LO(RTC_ICTL);
  R0 = ALARM(Z);
  W[P0] = R0.L;  /* enable Alarm interrupt */

  P0.L = LO(RTC_ISTAT);
  R0 = 0x807F(Z);
  W[P0] = R0.L;  /* clear any pending RTC interrupts */

  R0 = WRITE_COMPLETE(Z);
  Poll_WC1:  R1 = W[P0](Z);
             R1 = R1 & R0;  /* wait for Write Complete */
             CC = AZ;
             IF CC JUMP Poll_WC1;

/* RTC now running with correct RTC status */
GoToHibernate:
  P0.H = HI(VR_CTL);
  P0.L = LO(VR_CTL);
  R0 = W[P0](Z);
  BITCLR(R0, 0);   /* Clear FREQ (bits 0 and 1) to */
  BITCLR(R0, 1);   /* go to Hibernate State        */
  BITSET(R0, BITPOS(WAKE));  /* Enable RTC Wakeup  */
```

```
W[P0] = R0.L;

CLI R0;  /* Use PLL programming sequence to */
IDLE;    /* make VR_CTL changes take effect */
RTS;     /* Should Never Execute This       */
```

Unique Information for the ADSP-BF52x Processor

None.

15 PARALLEL PERIPHERAL INTERFACE

This chapter describes the parallel peripheral interface (PPI). Following an overview and a list of key features are a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF52x

For details regarding the number of PPIs for the ADSP-BF52x product, please refer to the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet*.

For PPI DMA channel assignments, refer to Table 6-7 on page 6-106 in Chapter 6, "Direct Memory Access".

For PPI interrupt vector assignments, refer to Table 5-3 on page 5-18 in Chapter 5, "System Interrupts".

To determine how each of the PPIs is multiplexed with other functional pins, refer to Table 9-2 on page 9-5 through Table 9-5 on page 9-9 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for each PPI, refer to Appendix A, "System MMR Assignments".

PPI behavior for the ADSP-BF52x that differs from the general information in this chapter can be found in the section "Unique Behavior for the ADSP-BF52x Processor" on page 15-39

Overview

The PPI is a half-duplex, bidirectional port accommodating up to 16 bits of data. It has a dedicated clock pin and three multiplexed frame sync pins. The highest system throughput is achieved with 8-bit data, since two 8-bit data samples can be packed as a single 16-bit word. In such a case, the earlier sample is placed in the 8 least significant bits (LSBs).

Features

The PPI includes these features:

- Half duplex, bidirectional parallel port
- Supports up to 16 bits of data
- Programmable clock and frame sync polarities
- ITU-R 656 support
- Interrupt generation on overflow and underrun

Typical peripheral devices that can be interfaced to the PPI port:

- A/D converters
- D/A converters
- LCD panels
- CMOS sensors
- Video encoders
- Video decoders

Interface Overview

Figure 15-1 shows a block diagram of the PPI.

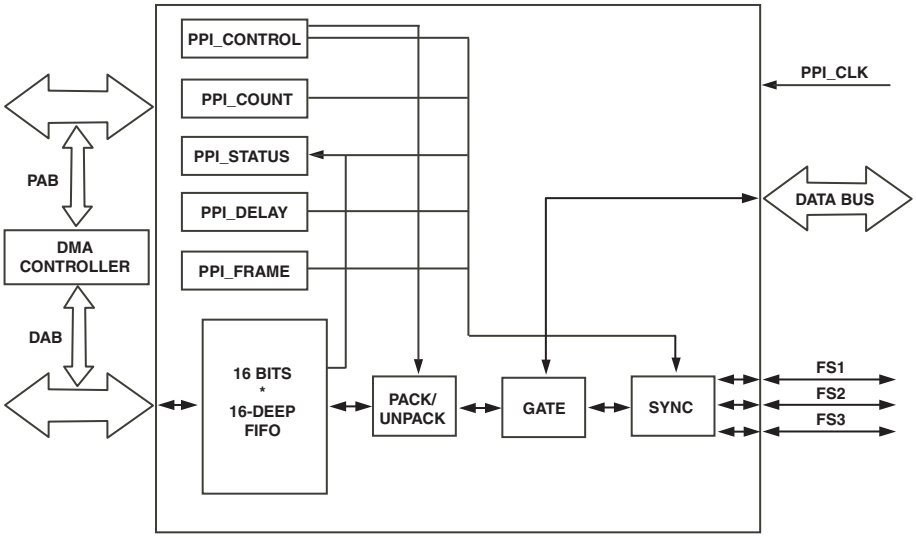


Figure 15-1. PPI Block Diagram

The PPI_CLK pin accepts an external clock input. It cannot source a clock internally.

When the PPI_CLK is not free-running, there may be additional latency cycles before data gets received or transmitted. In RX and TX modes, there may be at least 2 cycles latency before valid data is received or transmitted.

The PPI_CLK not only supplies the PPI module itself, but it also can clock one or more GP Timers to work synchronously with the PPI. Depending on PPI operation mode, the PPI_CLK can either equal or invert the TMRCLK input. For more information, see Chapter 10, “General-Purpose Timers”.

Description of Operation

Table 15-1 shows all the possible modes of operation for the PPI.

Table 15-1. PPI Possible Operating Modes

PPI Mode	# of Syncs	PORT_DIR	PORT_CFG	XFR_TYPE	POLC	POLS	FLD_ SEL
RX mode, 0 frame syncs, external trigger	0	0	11	11	0 or 1	0 or 1	0
RX mode, 0 frame syncs, internal trigger	0	0	11	11	0 or 1	0 or 1	1
RX mode, 1 external frame sync	1	0	00	11	0 or 1	0 or 1	0
RX mode, 2 or 3 external frame syncs	3	0	10	11	0 or 1	0 or 1	0
RX mode, 2 or 3 internal frame syncs	3	0	01	11	0 or 1	0 or 1	0
RX mode, ITU-R 656, active field only	embedded	0	00	00	0 or 1	0	0 or 1
RX mode, ITU-R 656, vertical blanking only	embedded	0	00	10	0 or 1	0	0
RX mode, ITU-R 656, entire field	embedded	0	00	01	0 or 1	0	0
TX mode, 0 frame syncs	0	1	00	00	0 or 1	0 or 1	0
TX mode, 1 internal or external frame sync	1	1	00	11	0 or 1	0 or 1	0
TX mode, 2 external frame syncs	2	1	01	11	0 or 1	0 or 1	0

Table 15-1. PPI Possible Operating Modes (Continued)

PPI Mode	# of Syncs	PORT_DIR	PORT_CFG	XFR_TYPE	POLC	POLS	FLD_ SEL
TX mode, 2 or 3 internal frame syncs, FS3 sync'd to FS1 assertion	3	1	01	11	0 or 1	0 or 1	0
TX mode, 2 or 3 internal frame syncs, FS3 sync'd to FS2 assertion	3	1	11	11	0 or 1	0 or 1	0

Functional Description

The following sections describe the function of the PPI.

ITU-R 656 Modes

The PPI supports three input modes for ITU-R 656-framed data. These modes are described in this section. Although the PPI does not explicitly support an ITU-R 656 output mode, recommendations for using the PPI for this situation are provided as well.

ITU-R 656 Background

According to the ITU-R 656 recommendation (formerly known as CCIR-656), a digital video stream has the characteristics shown in Figure 15-2, and Figure 15-3 for 525/60 (NTSC) and 625/50 (PAL) systems. The processor supports only the bit-parallel mode of ITU-R 656. Both 8- and 10-bit video element widths are supported.

In this mode, the horizontal (H), vertical (V), and field (F) signals are sent as an embedded part of the video datastream in a series of bytes that form a control word. The start of active video (SAV) and end of active video (EAV) signals indicate the beginning and end of data elements to read in on each line. SAV occurs on a 1-to-0 transition of H, and EAV begins on a

0-to-1 transition of H. An entire field of video is comprised of active video + horizontal blanking (the space between an EAV and SAV code) and vertical blanking (the space where V = 1). A field of video commences on a transition of the F bit. The “odd field” is denoted by a value of F = 0, whereas F = 1 denotes an even field. Progressive video makes no distinction between field 1 and field 2, whereas interlaced video requires each field to be handled uniquely, because alternate rows of each field combine to create the actual video image.

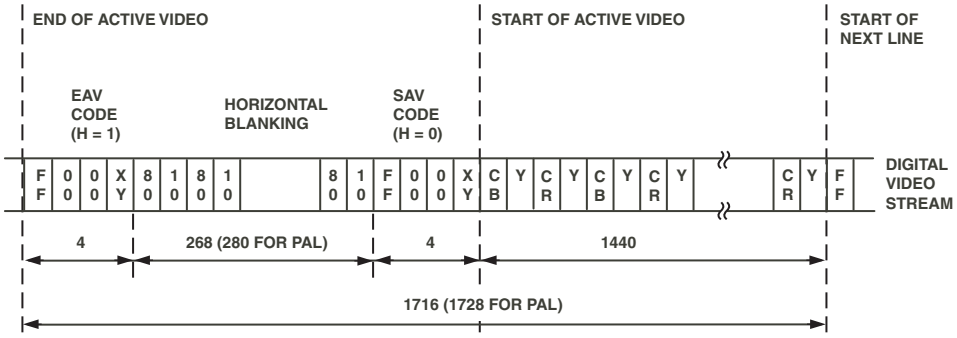
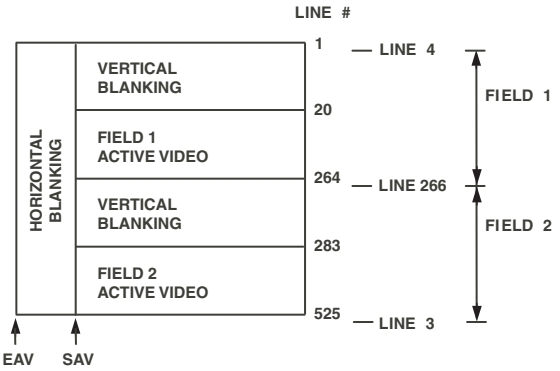
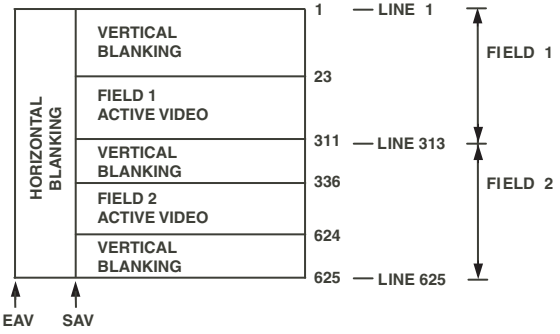

END OF ACTIVE VIDEO
START OF ACTIVE VIDEO
START OF NEXT LINE
EAV CODE (H = 1)
HORIZONTAL BLANKING
SAV CODE (H = 0)
FF 00 00 XY 80 10 80 10
80 10 FF 00 00 XY CB Y CR Y CB Y CR Y
CR Y FF
DIGITAL VIDEO STREAM
4
268 (280 FOR PAL)
4
1440
1716 (1728 FOR PAL)

Figure 15-2. ITU-R 656 8-Bit Parallel Data Stream for NTSC (PAL) Systems



LINE NUMBER	F	V	H (EAV)	H (SAV)
1-3, 266-282	1	1	1	0
4-19, 264-265	0	1	1	0
20-263	0	0	1	0
283-525	1	0	1	0



LINE NUMBER	F	V	H (EAV)	H (SAV)
1-22, 311-312	0	1	1	0
23-310	0	0	1	0
313-335, 624-625	1	1	1	0
336-623	1	0	1	0

Figure 15-3. Typical Video Frame Partitioning for NTSC/PAL Systems for ITU-R BT.656-4

The SAV and EAV codes are shown in more detail in Table 15-2. Note there is a defined preamble of three bytes (0xFF, 0x00, 0x00), followed by the XY status word, which, aside from the F (field), V (vertical blanking) and H (horizontal blanking) bits, contains four protection bits for single-bit error detection and correction. Note F and V are only allowed to change as part of EAV sequences (that is, transition from H = 0 to H = 1). The bit definitions are as follows:

- F = 0 for field 1
- F = 1 for field 2

- V = 1 during vertical blanking
- V = 0 when not in vertical blanking
- H = 0 at SAV
- H = 1 at EAV
- P3 = V XOR H
- P2 = F XOR H
- P1 = F XOR V
- P0 = F XOR V XOR H

In many applications, video streams other than the standard NTSC/PAL formats (for example, CIF, QCIF) can be employed. Because of this, the processor interface is flexible enough to accommodate different row and field lengths. In general, as long as the incoming video has the proper EAV/SAV codes, the PPI can read it in. In other words, a CIF image could be formatted to be "656-compliant," where EAV and SAV values define the range of the image for each line, and the V and F codes can be used to delimit fields and frames.

Table 15-2. Control Byte Sequences for 8-bit and 10-bit ITU-R 656 Video

	8-bit Data								**10-bit Data**	
	D9 (MSB)	D8	D7	D6	D5	D4	D3	D2	D1	D0
Preamble	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
Control Byte	1	F	V	H	P3	P2	P1	P0	0	0

ITU-R 656 Input Modes

Figure 15-4 shows a general illustration of data movement in the ITU-R 656 input modes. In the figure, the clock `CLK` is either provided by the video source or supplied externally by the system.

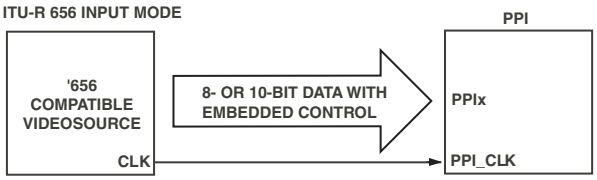


Figure 15-4. ITU-R 656 Input Modes

There are three submodes supported for ITU-R 656 inputs: entire field, active video only, and vertical blanking interval only. Figure 15-5 shows these three submodes.

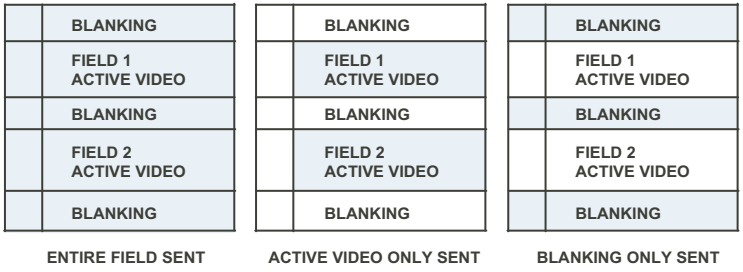


Figure 15-5. ITU-R 656 Input Submodes

Entire Field

In this mode, the entire incoming bitstream is read in through the PPI. This includes active video as well as control byte sequences and ancillary data that may be embedded in horizontal and vertical blanking intervals. Data transfer starts immediately after synchronization to field 1 occurs,

but does not include the first EAV code that contains the F = 0 assignment.

Note the first line transferred in after enabling the PPI will be missing its first 4-byte preamble. However, subsequent lines and frames should have all control codes intact.

One side benefit of this mode is that it enables a "loopback" feature through which a frame or two of data can be read in through the PPI and subsequently output to a compatible video display device. Of course, this requires multiplexing on the PPI pins, but it enables a convenient way to verify that 656 data can be read into and written out from the PPI.

Active Video Only

This mode is used when only the active video portion of a field is of interest, and not any of the blanking intervals. The PPI ignores (does not read in) all data between EAV and SAV, as well as all data present when V = 1. In this mode, the control byte sequences are not stored to memory; they are filtered out by the PPI. After synchronizing to the start of field 1, the PPI ignores incoming samples until it sees an SAV.

In this mode, the user specifies the number of total (active plus vertical blanking) lines per frame in the PPI_FRAME MMR.

Vertical Blanking Interval (VBI) only

In this mode, data transfer is only active while V = 1 is in the control byte sequence. This indicates that the video source is in the midst of the vertical blanking interval (VBI), which is sometimes used for ancillary data transmission. The ITU-R 656 recommendation specifies the format for these ancillary data packets, but the PPI is not equipped to decode the packets themselves. This task must be handled in software. Horizontal blanking data is logged where it coincides with the rows of the VBI. Control byte sequence information is always logged. The user specifies the number of total lines (active plus vertical blanking) per frame in the PPI_FRAME MMR.

Note the VBI is split into two regions within each field. From the PPI's standpoint, it considers these two separate regions as one contiguous space. However, keep in mind that frame synchronization begins at the start of field 1, which doesn't necessarily correspond to the start of vertical blanking. For instance, in 525/60 systems, the start of field 1 (F = 0) corresponds to line 4 of the VBI.

ITU-R 656 Output Mode

The PPI does not explicitly provide functionality for framing an ITU-R 656 output stream with proper preambles and blanking intervals. However, with the TX mode with 0 frame syncs, this process can be supported manually. Essentially, this mode provides a streaming operation from memory out through the PPI. Data and control codes can be set up in memory prior to sending out the video stream. With the 2D DMA engine, this could be performed in a number of ways. For instance, one line of blanking (H + V) could be stored in a buffer and sent out N times by the DMA controller when appropriate, before proceeding to DMA active video. Alternatively, one entire field (with control codes and blanking) can be set up statically in a buffer while the DMA engine transfers only the active video region into the buffer, on a frame-by-frame basis.

Frame Synchronization in ITU-R 656 Modes

Synchronization in ITU-R 656 modes always occurs at the falling edge of F, the field indicator. This corresponds to the start of field 1. Consequently, up to two fields might be ignored (for example, if field 1 just started before the PPI-to-camera channel was established) before data is received into the PPI.

Because all H and V signaling is embedded in the datastream in ITU-R 656 modes, the PPI_COUNT register is not necessary. However, the PPI_FRAME register is used in order to check for synchronization errors. The user programs this MMR for the number of lines expected in each frame of video, and the PPI keeps track of the number of EAV-to-SAV transitions that

occur from the start of a frame until it decodes the end-of-frame condition (transition from F = 1 to F = 0). At this time, the actual number of lines processed is compared against the value in PPI_FRAME. If there is a mismatch, the FT_ERR bit in the PPI_STATUS register is asserted. For instance, if an SAV transition is missed, the current field will only have NUM_ROWS - 1 rows, but resynchronization will reoccur at the start of the next frame.

Upon completing reception of an entire field, the field status bit is toggled in the PPI_STATUS register. This way, an interrupt service routine (ISR) can discern which field was just read in.

General-Purpose PPI Modes

The general-purpose PPI modes are intended to suit a wide variety of data capture and transmission applications. Table 15-3 summarizes these modes. If a particular mode shows a given PPI_FSx frame sync not being used, this implies that the pin is available for its alternate, multiplexed functions.

Table 15-3. General-Purpose PPI Modes

GP PPI Mode	PPI_FS1 Direction	PPI_FS2 Direction	PPI_FS3 Direction	Data Direction
RX mode, 0 frame syncs, external trigger	Input	Not used	Not used	Input
RX mode, 0 frame syncs, internal trigger	Not used	Not used	Not used	Input
RX mode, 1 external frame sync	Input	Not used	Not used	Input
RX mode, 2 or 3 external frame syncs	Input	Input	Input (if used)	Input
RX mode, 2 or 3 internal frame syncs	Output	Output	Output (if used)	Input
TX mode, 0 frame syncs	Not used	Not used	Not used	Output
TX mode, 1 external frame sync	Input	Not used	Not used	Output

Table 15-3. General-Purpose PPI Modes (Continued)

GP PPI Mode	PPI_FS1 Direction	PPI_FS2 Direction	PPI_FS3 Direction	Data Direction
TX mode, 2 external frame syncs	Input	Input	Not used	Output
TX mode, 1 internal frame sync	Output	Not used	Not used	Output
TX mode, 2 or 3 internal frame syncs	Output	Output	Output (if used)	Output

Figure 15-6 illustrates the general flow of the general purpose PPI modes. The top of the diagram shows an example of RX mode with one external frame sync. After the PPI receives the hardware frame sync pulse (PPI_FS1), it delays for the duration of the PPI_CLK cycles programmed into PPI_DELAY. The DMA controller then transfers in the number of samples specified by PPI_COUNT. Every sample that arrives after this, but before the next PPI_FS1 frame sync arrives, is ignored and not transferred onto the DMA bus.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been read in, the sample counter reinitializes to 0 and starts to count up to PPI_COUNT again. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

The bottom of Figure 15-6 shows an example of TX mode, one internal frame sync. After PPI_FS1 is asserted, there is a latency of one PPI_CLK cycle, and then there is a delay for the number of PPI_CLK cycles programmed into PPI_DELAY. Next, the DMA controller transfers out the

number of samples specified by PPI_COUNT. No further DMA takes place until the next PPI_FS1 sync and programmed delay occur.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been transferred out, the sync has priority and starts a new line transfer sequence. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

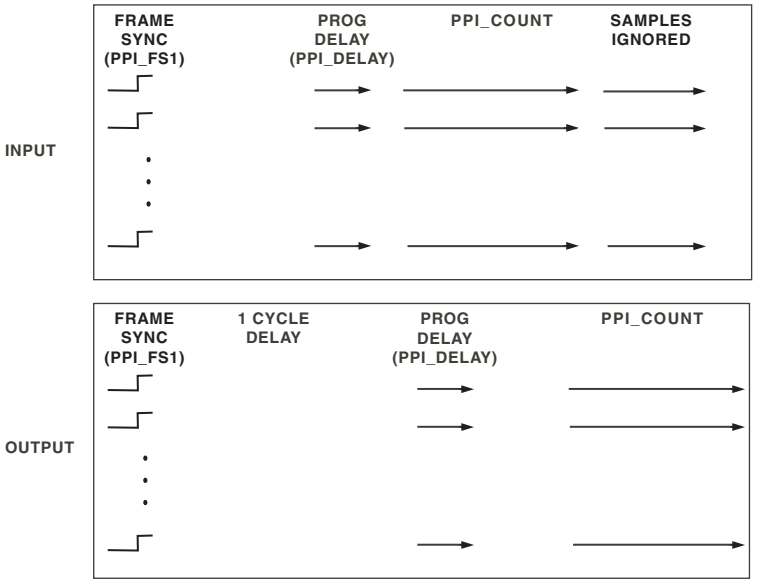


Figure 15-6. General Flow for GP Modes (Assumes Positive Assertion of PPI_FS1)

Data Input (RX) Modes

The PPI supports several modes for data input. These modes differ chiefly by the way the data is framed. Refer to Table 15-1 on page 15-4 for information on how to configure the PPI for each mode.

No Frame Syncs

These modes cover the set of applications where periodic frame syncs are not generated to frame the incoming data. There are two options for starting the data transfer, both configured by the PPI_CONTROL register.

- External trigger: An external source sends a single frame sync (tied to PPI_FS1) at the start of the transaction, when FLD_SEL = 0 and PORT_CFG = b#11.
- Internal trigger: Software initiates the process by setting PORT_EN = 1 with FLD_SEL = 1 and PORT_CFG = b#11.

All subsequent data manipulation is handled via DMA. For example, an arrangement could be set up between alternating 1K byte memory buffers. When one fills up, DMA continues with the second buffer, at the same time that another DMA operation is clearing the first memory buffer for reuse.

Due to clock domain synchronization in RX modes with no frame syncs, there may be a delay of at least two PPI_CLK cycles between when the mode is enabled and when valid data is received. Therefore, detection of the start of valid data should be managed by software.

1, 2, or 3 External Frame Syncs

The frame syncs are level-sensitive signals. The 1-sync mode is intended for analog-to-digital converter (ADC) applications. The top part of Figure 15-7 shows a typical illustration of the system setup for this mode.

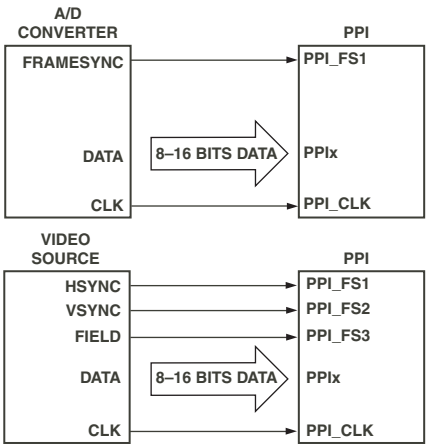


Figure 15-7. RX Mode, External Frame Syncs

The 3-sync mode shown at the bottom of Figure 15-7 supports video applications that use hardware signaling (`HSYNC`, `VSYNC`, `FIELD`) in accordance with the ITU-R 601 recommendation. The mapping for the frame syncs in this mode is `PPI_FS1 = HSYNC`, `PPI_FS2 = VSYNC`, `PPI_FS3 = FIELD`. Please refer to "Frame Synchronization in GP Modes" on page 15-20 for more information about frame syncs in this mode.

A 2-sync mode is supported by not enabling the `PPI_FS3` pin. See the *Product Specific Implementation* section for information on how this is achieved on this processor.

2 or 3 Internal Frame Syncs

This mode can be useful for interfacing to video sources that can be slaved to a master processor. In other words, the processor controls when to read from the video source by asserting `PPI_FS1` and `PPI_FS2`, and then reading

data into the PPI. The PPI_FS3 frame sync provides an indication of which field is currently being transferred, but since it is an output, it can simply be left floating if not used. Figure 15-8 shows a sample application for this mode.

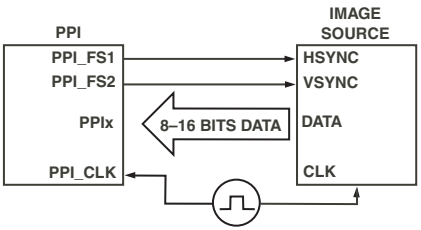


Figure 15-8. RX Mode, Internal Frame Syncs

Data Output (TX) Modes

The PPI supports several modes for data output. These modes differ chiefly by the way the data is framed. Refer to Table 15-1 on page 15-4 for information on how to configure the PPI for each mode.

No Frame Syncs

In this mode, data blocks specified by the DMA controller are sent out through the PPI with no framing. That is, once the DMA channel is configured and enabled, and the PPI is configured and enabled, data transfers

will take place immediately, synchronized to PPI_CLK. See Figure 15-9 for an illustration of this mode.

In this mode, there is a delay of up to 16 SCLK cycles (for > 8-bit data) or 32 SCLK cycles (for 8-bit data) between enabling the PPI and transmission of valid data. Furthermore, DMA must be configured to transmit at least 16 samples (for > 8-bit data) or 32 samples (for 8-bit data).

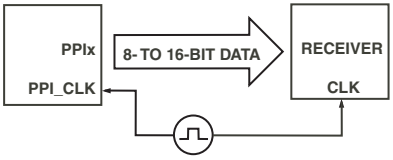


Figure 15-9. TX Mode, 0 Frame Syncs

1 or 2 External Frame Syncs

In these modes, an external receiver can frame data sent from the PPI. Both 1-sync and 2-sync modes are supported. The top diagram in

Figure 15-10 shows the 1-sync case, while the bottom diagram illustrates the 2-sync mode.

There is a mandatory delay of 1.5 PPI_CLK cycles, plus the value programmed in PPI_DELAY, between assertion of the external frame sync(s) and the transfer of valid data out through the PPI.

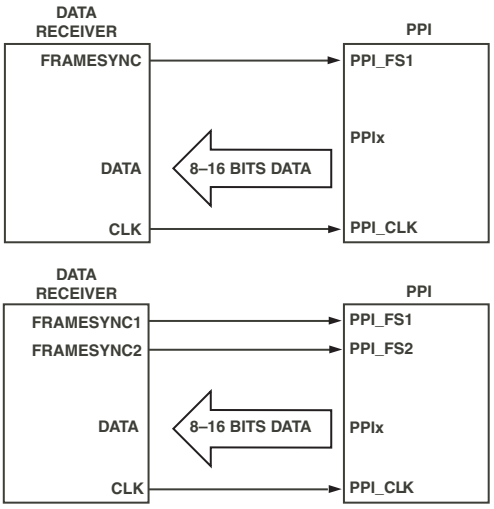


Figure 15-10. TX Mode, 1 or 2 External Frame Syncs

1, 2, or 3 Internal Frame Syncs

The 1-sync mode is intended for interfacing to digital-to-analog converters (DACs) with a single frame sync. The top part of Figure 15-11 shows an example of this type of connection.

The 3-sync mode is useful for connecting to video and graphics displays, as shown in the bottom part of Figure 15-11. A 2-sync mode is implicitly supported by leaving PPI_FS3 unconnected in this case.

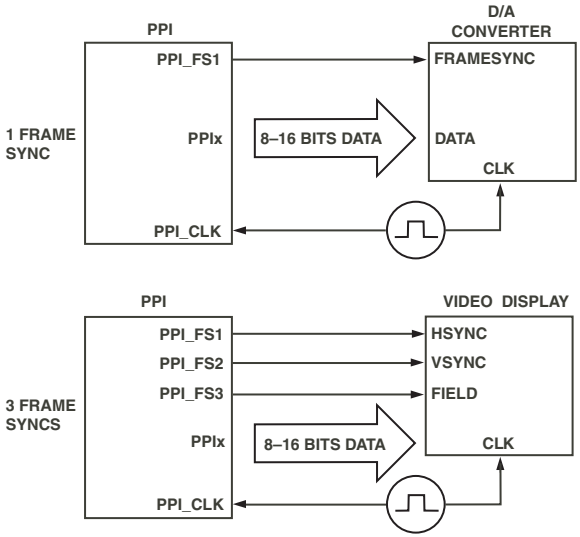


Figure 15-11. PPI GP Output

Frame Synchronization in GP Modes

Frame synchronization in general purpose modes operates differently in modes with internal frame syncs than in modes with external frame syncs.

Modes With Internal Frame Syncs

In modes with internal frame syncs, `PPI_FS1` and `PPI_FS2` link directly to the pulsewidth modulation (PWM) circuits of general purpose timers. See the Chapter 10, “General-Purpose Timers” for information on how this is achieved on this processor. This allows for arbitrary pulse widths and periods to be programmed for these signals using the existing `TIMERx` registers. This capability accommodates a wide range of timing needs. Note these PWM circuits are clocked by `PPI_CLK`, not by `SCLK` (as during conventional timer PWM operation). If `PPI_FS2` is not used in the configured PPI mode, its corresponding timer operates as it normally would, unrestricted in functionality. The state of `PPI_FS3` depends completely on the

state of PPI_FS1 and/or PPI_FS2, so PPI_FS3 has no inherent programmability.

To program PPI_FS1 and/or PPI_FS2 for operation in an internal frame sync mode:

1. Configure and enable DMA for the PPI. See "DMA Operation" on page 15-23.
2. Configure the width and period for each frame sync signal via the appropriate TIMER_WIDTH and TIMER_PERIOD registers.
3. Set up the appropriate TIMER_CONFIG register(s) for PWM_OUT mode. This includes setting CLK_SEL to 1 and TIN_SEL to 1 for each timer involved.
4. Write to PPI_CONTROL to configure and enable the PPI.
5. Write to TIMER_ENABLE to enable the appropriate timer(s).

It is important to guarantee proper frame sync polarity between the PPI and timer peripherals. To do this, make sure that if PPI_CONTROL[15:14] = b#10 or b#11, the PULSE_HI bit is cleared in the appropriate TIMER_CONFIG register(s). Likewise, if PPI_CONTROL[15:14] = b#00 or b#01, the PULSE_HI bit should be set in the appropriate TIMER_CONFIG register(s).

To switch to another PPI mode not involving internal frame syncs:

1. Disable the PPI (using PPI_CONTROL).
2. Disable the appropriate timer(s) (using TIMER_DISABLE).

Modes With External Frame Syncs

In RX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins become edge-sensitive inputs. In such modes the timers associated with the PPI_FS1 and PPI_FS2 pins can still be used for a purpose not involving the actual pin. However, timer access to a TMRx pin is disabled when the

PPI is using that pin for a PPI_FSx frame sync input function. For modes that do not require PPI_FS2, the associated timer is not restricted in functionality and can be operated as if the PPI were not being used (that is, the TMR1 pin becomes available for timer use as well). For more information on configuring and using the timers, please refer to the *General-Purpose Timers* chapter.

In RX mode with 3 external frame syncs, the start of frame detection occurs where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This happens at the start of field 1. Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

In TX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins are treated as edge-sensitive inputs. In this mode, it is not necessary to configure the timer(s) associated with the frame sync(s) as input(s), or to enable them via the TIMER_ENABLE register. Additionally, the actual timers themselves are available for use, even though the timer pin(s) are taken over by the PPI. In this case, there is no requirement that the timebase (configured by TIN_SEL in TIMERx_CONFIG) be PPI_CLK.

However, if using a timer whose pin is connected to an external frame sync, be sure to disable the pin via the OUT_DIS bit in TIMER_CONFIG. Then the timer itself can be configured and enabled for non-PPI use without affecting PPI operation in this mode. For more information, see the *General-Purpose Timers* chapter.

Programming Model

The following sections describe the PPI programming model.

DMA Operation

The PPI must be used with the processor's DMA engine. This section discusses how the two interact. For additional information about the DMA engine, including explanations of DMA registers and DMA operations, please refer to the *Direct Memory Access* chapter.

The PPI DMA channel can be configured for either transmit or receive operation, and it has a maximum throughput of (PPI_CLK) × (16 bits/transfer). In modes where data lengths are greater than eight bits, only one element can be clocked in per PPI_CLK cycle, and this results in reduced bandwidth (since no packing is possible). The highest throughput is achieved with 8-bit data and PACK_EN = 1 (packing mode enabled). Note for 16-bit packing mode, there must be an even number of data elements.

Configuring the PPI's DMA channel is a necessary step toward using the PPI interface. It is the DMA engine that generates interrupts upon completion of a row, frame, or partial-frame transfer. It is also the DMA engine that coordinates the origination or destination point for the data that is transferred through the PPI.

The processor's 2D DMA capability allows the processor to be interrupted at the end of a line or after a frame of video has been transferred, as well as if a DMA error occurs. In fact, the specification of the DMA_XCOUNT and DMA_YCOUNT MMRs allows for flexible data interrupt points. For example, assume the DMA registers XMODIFY = YMODIFY = 1. Then, if a data frame

contains 320 x 240 bytes (240 rows of 320 bytes each), these conditions hold:

- Setting XCOUNT = 320, YCOUNT = 240, and DI_SEL = 1 (the DI_SEL bit is located in DMA_CONFIG) interrupts on every row transferred, for the entire frame.
- Setting XCOUNT = 320, YCOUNT = 240, and DI_SEL = 0 interrupts only on the completion of the frame (when 240 rows of 320 bytes have been transferred).
- Setting XCOUNT = 38,400 (320 x 120), YCOUNT = 2, and DI_SEL = 1 causes an interrupt when half of the frame has been transferred, and again when the whole frame has been transferred.

The general procedure for setting up DMA operation with the PPI follows.

1. Configure DMA registers as appropriate for desired DMA operating mode.
2. Enable the DMA channel for operation.
3. Configure appropriate PPI registers.
4. Enable the PPI by writing a 1 to bit 0 in PPI_CONTROL.
5. If internally generated frame syncs are used, write to the TIMER_ENABLE register to enable the timers linked to the PPI frame syncs.

Figure 15-12 shows a flow diagram detailing the steps on how to configure the PPI for the various modes of operation.

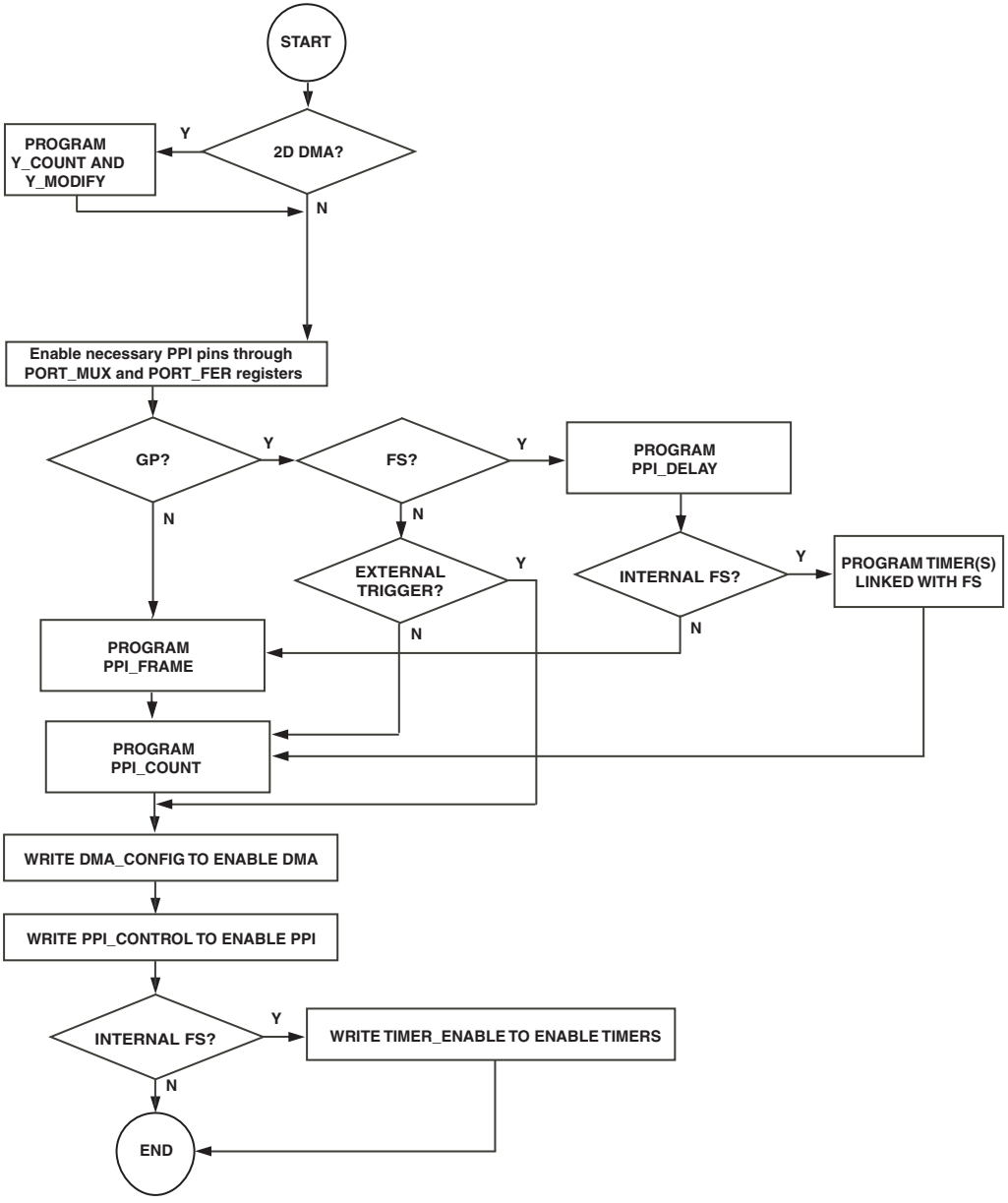


Figure 15-12. PPI Flow Diagram

PPI Registers

The PPI has five memory-mapped registers (MMRs) that regulate its operation. These registers are the PPI control register (PPI_CONTROL), the PPI status register (PPI_STATUS), the delay count register (PPI_DELAY), the transfer count register (PPI_COUNT), and the lines per frame register (PPI_FRAME).

Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

PPI Control Register (PPI_CONTROL)

The PPI_CONTROL register configures the PPI for operating mode, control signal polarities, and data width of the port. See Figure 15-13 for a bit diagram of this MMR.

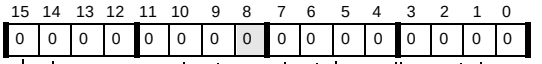
The POLC and POLS bits allow for selective signal inversion of the PPI_CLK and PPI_FS1/PPI_FS2 signals, respectively. This provides a mechanism to connect to data sources and receivers with a wide array of control signal polarities. Often, the remote data source/receiver also offers configurable signal polarities, so the POLC and POLS bits simply add increased flexibility.

The DLEN[2:0] field is programmed to specify the width of the PPI port in any mode. Note any width from 8 to 16 bits is supported, with the exception of a 9-bit port width. Any pins unused by the PPI as a result of the DLEN setting are free for use in their other functions.

In ITU-R 656 modes, the DLEN field should not be configured for anything greater than a 10-bit port width. If it is, the PPI will reserve extra pins, making them unusable by other peripherals.

The SKIP_EN bit, when set, enables the selective skipping of data elements being read in through the PPI. By ignoring data elements, the PPI is able to conserve DMA bandwidth.

PPI Control Register (PPI_CONTROL)



POLS
0 - PPI_FS1 and PPI_FS2 are treated as rising edge asserted
1 - PPI_FS1 and PPI_FS2 are treated as falling edge asserted

POLC
0 - PPI samples data on rising edge and drives data on falling edge of PPI_CLK
1 - PPI samples data on falling edge and drives data on rising edge of PPI_CLK

DLEN[2:0] (Data Length)
000 - 8 bits
001 - 10 bits
010 - 11 bits
011 - 12 bits
100 - 13 bits
101 - 14 bits
110 - 15 bits
111 - 16 bits

SKIP_EO (Skip Even Odd)
In ITU-R 656 and GP Input modes:
0 - Skip odd-numbered elements
1 - Skip even-numbered elements

SKIP_EN (Skip Enable)
In ITU-R 656 and GP Input modes:
0 - Skipping disabled
1 - Skipping enabled

PACK_EN (Packing Mode Enable)
0 - Disabled
1 - Output mode, unpacking enabled; Input mode, packing enabled

FLD_SEL (Active Field Select)
In ITU-R 656 modes, when XFR_TYPE = 00:
0 - Field 1
1 - Fields 1 and 2
In RX mode with external frame sync, when PORT_CFG = 11:
0 - External trigger
1 - Internal trigger

PORT_EN (Enable)
0 - PPI disabled
1 - PPI enabled

PORT_DIR (Direction)
0 - PPI in Receive mode (input)
1 - PPI in Transmit mode (output)

XFR_TYPE[1:0] (Transfer Type)
In Input mode:
00 - ITU-R 656, Active Field Only
01 - ITU-R 656, Entire Field
10 - ITU-R 656, Vertical Blanking Only
11 - Non-ITU-R 656 mode
In Output mode:
00, 01, 10 - Sync-less Output mode
11 - Output mode with 1, 2, or 3 frame syncs

PORT_CFG[1:0] (Port Configuration)
In non-ITU-R 656 Input modes (PORT_DIR = 0, XFR_TYPE = 11):
00 - 1 external frame sync
01 - 2 or 3 internal frame syncs
10 - 2 or 3 external frame syncs
11 - 0 frame syncs, triggered
In Output modes with frame syncs (PORT_DIR = 1, XFR_TYPE = 11):
00 - 1 frame sync
01 - 2 or 3 frame syncs
10 - Reserved
11 - Sync PPI_FS3 to assertion of PPI_FS2 rather than of PPI_FS1.

Figure 15-13. PPI Control Register

When the SKIP_EN bit is set, the SKIP_EO bit allows the PPI to ignore either the odd or the even elements in an input datastream. This is useful, for instance, when reading in a color video signal in YCbCr format (Cb, Y, Cr, Y, Cb, Y, Cr, Y...). Skipping every other element allows the PPI to only read in the luma (Y) or chroma (Cr or Cb) values. This could also be useful when synchronizing two processors to the same incoming video stream. One processor could handle luma processing and the other (whose SKIP_EO bit is set differently from the first processor's) could handle chroma processing. This skipping feature is valid in ITU-R 656 modes and RX modes with external frame syncs.

The PACK_EN bit only has meaning when the PPI port width (selected by DLEN[2:0]) is eight bits. Every PPI_CLK-initiated event on the DMA bus (that is, an input or output operation) handles 16-bit entities. In other words, an input port width of ten bits still results in a 16-bit input word for every PPI_CLK; the upper 6 bits are 0s. Likewise, a port width of eight bits also results in a 16-bit input word, with the upper eight bits all 0s. In the case of 8-bit data, it is usually more efficient to pack this information so that there are two bytes of data for every 16-bit word. This is the function of the PACK_EN bit. When set, it enables packing for all RX modes.

Consider this data transported into the PPI via DMA:
`0xCE, 0xFA, 0xFE, 0xCA....`

- With `PACK_EN` set:

  This is read into the PPI, configured for an 8-bit port width:
`0xCE, 0xFA, 0xFE, 0xCA...`

- This is transferred onto the DMA bus:
`0xFACE, 0xCAFE,...`

- With `PACK_EN` cleared:

  This is read into the PPI:
`0xCE, 0xFA, 0xFE, 0xCA,...`

- This is transferred onto the DMA bus:
`0x00CE, 0x00FA, 0x00FE, 0x00CA,...`

For TX modes, setting `PACK_EN` enables unpacking of bytes. Consider this data in memory, to be transported out through the PPI via DMA:
`0xFACE CAFE....`

(0xFA and 0xCA are the two most significant bits (MSBs) of their respective 16-bit words)

- With PACK_EN set:

  This is DMAed to the PPI:

```
0xFACE, 0xCAFE,...
```

- This is transferred out through the PPI, configured for an 8-bit port width (note LSBs are transferred first):

```
0xCE, 0xFA, 0xFE, 0xCA,...
```

- With PACK_EN cleared:

  This is DMAed to the PPI:

```
0xFACE, 0xCAFE,...
```

- This is transferred out through the PPI, configured for an 8-bit port width:

```
0xCE, 0xFE,...
```

The FLD_SEL bit is used primarily in the active field only ITU-R 656 mode. The FLD_SEL bit determines whether to transfer in only field 1 of each video frame, or both fields 1 and 2. Thus, it allows a savings in DMA bandwidth by transferring only every other field of active video.

The PORT_CFG[1:0] field is used to configure the operating mode of the PPI. It operates in conjunction with the PORT_DIR bit, which sets the direction of data transfer for the port. The XFR_TYPE[1:0] field is also used to configure operating mode and is discussed below. See Table 15-1 on page 15-4 for the possible operating modes for the PPI.

The XFR_TYPE[1:0] field configures the PPI for various modes of operation. Refer to Table 15-1 on page 15-4 to see how XFR_TYPE[1:0] interacts with other bits in PPI_CONTROL to determine the PPI operating mode.

The PORT_EN bit, when set, enables the PPI for operation.

When configured as an input port, the PPI does not start data transfer after being enabled until the appropriate synchronization signals are received. If configured as an output port, transfer (including the appropriate synchronization signals) begins as soon as the frame syncs (timer units) are enabled, so all frame syncs must be configured before this happens. Refer to the section "Frame Synchronization in GP Modes" on page 15-20 for more information.

PPI Status Register (PPI_STATUS)

The PPI_STATUS register, shown in Figure 15-14, contains bits that provide information about the current operating state of the PPI.

The ERR_DET bit is a sticky bit that denotes whether or not an error was detected in the ITU-R 656 control word preamble. The bit is valid only in

ITU-R 656 modes. If ERR_DET = 1, an error was detected in the preamble. If ERR_DET = 0, no error was detected in the preamble.

PPI Status Register (PPI_STATUS)

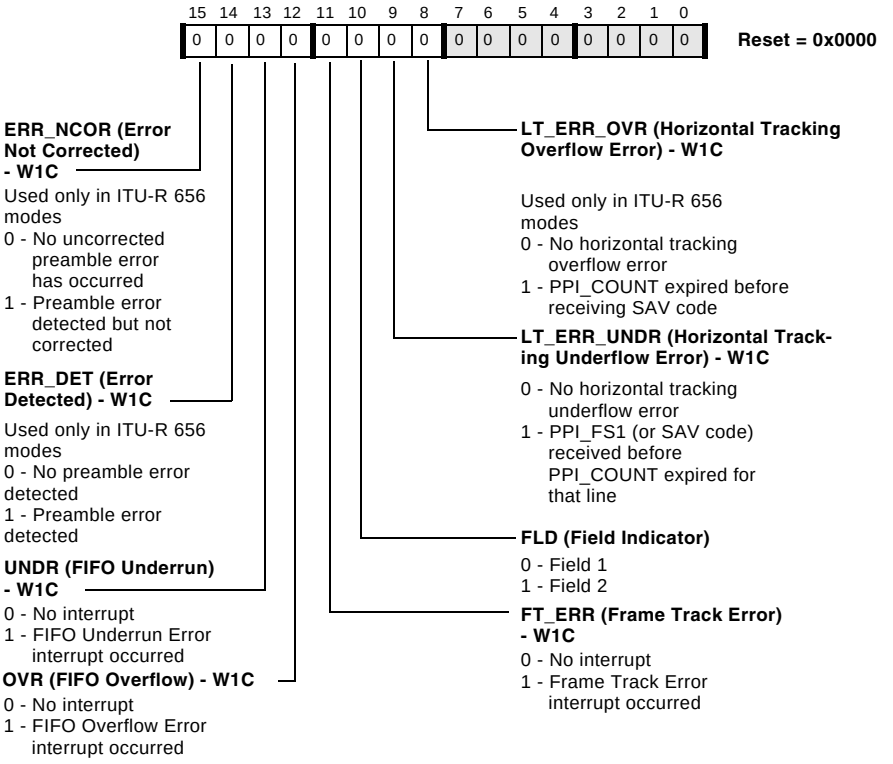

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
ERR_NCOR (Error Not Corrected) - W1C
Used only in ITU-R 656 modes
0 - No uncorrected preamble error has occurred
1 - Preamble error detected but not corrected
ERR_DET (Error Detected) - W1C
Used only in ITU-R 656 modes
0 - No preamble error detected
1 - Preamble error detected
UNDR (FIFO Underrun) - W1C
0 - No interrupt
1 - FIFO Underrun Error interrupt occurred
OVR (FIFO Overflow) - W1C
0 - No interrupt
1 - FIFO Overflow Error interrupt occurred
LT_ERR_OVR (Horizontal Tracking Overflow Error) - W1C
Used only in ITU-R 656 modes
0 - No horizontal tracking overflow error
1 - PPI_COUNT expired before receiving SAV code
LT_ERR_UNDR (Horizontal Tracking Underflow Error) - W1C
0 - No horizontal tracking underflow error
1 - PPI_FS1 (or SAV code) received before PPI_COUNT expired for that line
FLD (Field Indicator)
0 - Field 1
1 - Field 2
FT_ERR (Frame Track Error) - W1C
0 - No interrupt
1 - Frame Track Error interrupt occurred

Figure 15-14. PPI Status Register

The ERR_NCOR bit is sticky and is relevant only in ITU-R 656 modes. If ERR_NCOR = 0 and ERR_DET = 1, all preamble errors that have occurred have been corrected. If ERR_NCOR = 1, an error in the preamble was detected but not corrected. This situation generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The FT_ERR bit is sticky and indicates, when set, that a frame track error has occurred. In this condition, the programmed number of lines per

frame in PPI_FRAME does not match up with the “frame start detect” condition (see the information note on page 15-36). A frame track error generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The FLD bit is set or cleared at the same time as the change in state of F (in ITU-R 656 modes) or PPI_FS3 (in other RX modes). It is valid for input modes only. The state of FLD reflects the current state of the F or PPI_FS3 signals. In other words, the FLD bit always reflects the current video field being processed by the PPI.

The OVR bit is sticky and indicates, when set, that the PPI FIFO has overflowed and can accept no more data. A FIFO overflow error generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The PPI FIFO is 16 bits wide and has 16 entries.

The UNDR bit is sticky and indicates, when set, that the PPI FIFO has underrun and is data-starved. A FIFO underrun error generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The LT_ERR_OVR and LT_ERR_UNDR bits are sticky and indicate, when set, that a line track error has occurred. These bits are valid for RX modes with recurring frame syncs only. If one of these bits is set, the programmed number of samples in PPI_COUNT did not match up with the actual number of samples counted between assertions of PPI_FS1 (for general-purpose modes) or start of active video (SAV) codes (for ITU-R 656 modes). If the PPI error interrupt is enabled in the SIC_IMASK register, an interrupt request is generated when one of these bits is set.

The LT_ERR_OVR flag signifies that a horizontal tracking overflow has occurred, where the value in PPI_COUNT was reached before a new SAV code was received. This flag does not apply for non ITU-R 656 modes; in this case, once the value in PPI_COUNT is reached, the PPI simply stops counting until receiving the next PPI_FS1 frame sync.

The LT_ERR_UNDR flag signifies that a horizontal tracking underflow has occurred, where a new SAV code or PPI_FS1 assertion occurred before the value in PPI_COUNT was reached.

PPI Delay Count Register (PPI_DELAY)

The PPI_DELAY register, shown in Figure 15-15, can be used in all configurations except ITU-R 656 modes and GP modes with 0 frame syncs. It contains a count of how many PPI_CLK cycles to delay after assertion of PPI_FS1 before starting to read in or write out data.

Note in TX modes using at least one frame sync, there is a one-cycle delay beyond what is specified in the PPI_DELAY register.

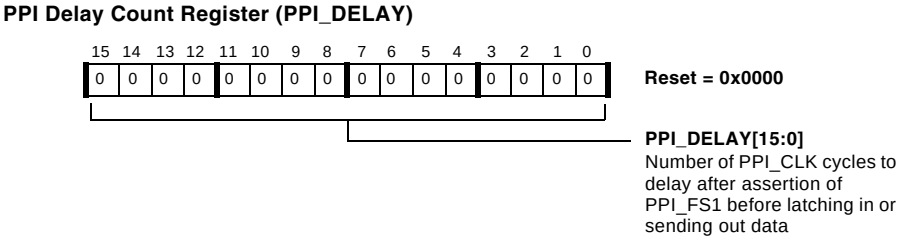

PPI Delay Count Register (PPI_DELAY)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
PPI_DELAY[15:0]
Number of PPI_CLK cycles to delay after assertion of PPI_FS1 before latching in or sending out data

Figure 15-15. PPI Delay Count Register

PPI Transfer Count Register (PPI_COUNT)

The PPI_COUNT register, shown in Figure 15-16, is used in all modes except “RX mode with 0 frame syncs, external trigger” and “TX mode with 0 frame syncs.” For RX modes, this register holds the number of samples to read into the PPI per line, minus one. For TX modes, it holds the number of samples to write out through the PPI per line, minus one. The register itself does not actually decrement with each transfer. Thus, at the beginning of a new line of data, there is no need to rewrite the value of

this register. For example, to receive or transmit 100 samples through the PPI, set PPI_COUNT to 99.

Take care to ensure that the number of samples programmed into PPI_COUNT is in keeping with the number of samples expected during the “horizontal” interval specified by PPI_FS1.

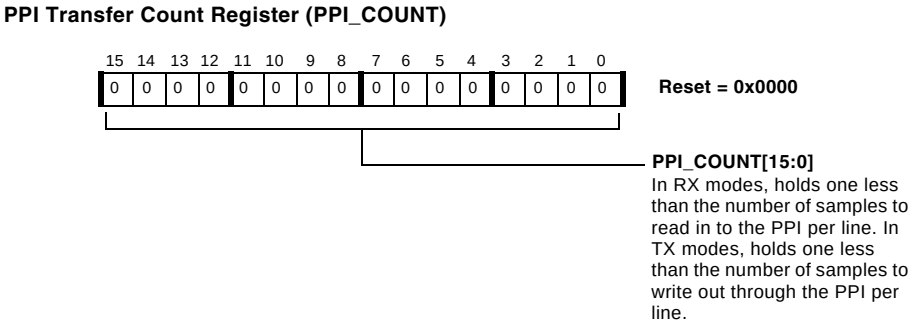

PPI Transfer Count Register (PPI_COUNT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
PPI_COUNT[15:0]
In RX modes, holds one less than the number of samples to read in to the PPI per line. In TX modes, holds one less than the number of samples to write out through the PPI per line.

Figure 15-16. PPI Transfer Count Register

PPI Lines Per Frame Register (PPI_FRAME)

The PPI_FRAME register, shown in Figure 15-17, is used in all TX and RX modes with two or three frame syncs. For ITU-R 656 modes, this register holds the number of lines expected per frame of data, where a frame is defined as field 1 and field 2 combined, designated by the F indicator in the ITU-R stream. Here, a line is defined as a complete ITU-R 656 SAV-EAV cycle.

For non ITU-R 656 modes with external frame syncs, a frame is defined as the data bounded between PPI_FS2 assertions, regardless of the state of PPI_FS3. A line is defined as a complete PPI_FS1 cycle. In these modes, PPI_FS3 is used only to determine the original “frame start” each time the PPI is enabled. It is ignored on every subsequent field and frame, and its state (high or low) is not important except during the original frame start.

If the start of a new frame (or field, for ITU-R 656 mode) is detected before the number of lines specified by PPI_FRAME have been transferred, a frame track error results, and the FT_ERR bit in PPI_STATUS is set. However, the PPI still automatically reinitializes to count to the value programmed in PPI_FRAME, and data transfer continues.

In ITU-R 656 modes, a frame start detect happens on the falling edge of F, the field indicator. This occurs at the start of field 1.

In RX mode with three external frame syncs, a frame start detect refers to a condition where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This occurs at the start of field 1. Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

When using RX mode with three external frame syncs, and only two syncs are needed, configure the PPI for 3-frame-sync operation and provide an external pull-down to GND for the PPI_FS3 pin.

PPI Lines Per Frame Register (PPI_FRAME)

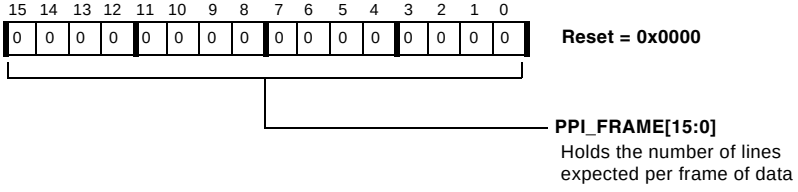

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
PPI_FRAME[15:0]
Holds the number of lines
expected per frame of data

Figure 15-17. PPI Lines Per Frame Register

Programming Examples

The PPI can be configured to receive data from a video source in several RX modes. The following programming examples (Listing 15-1 through Listing 15-5) describe the ITU-R 656 entire field input mode.

Listing 15-1. Configure DMA Registers

```
config_dma:
/*Assumes PPI is mapped to DMA channel 0.*/
  /* DMA0_START_ADDR */
  R0.L = rx_buffer;
  R0.H = rx_buffer;
  P0.L = lo(DMA0_START_ADDR);
  P0.H = hi(DMA0_START_ADDR);
  [P0] = R0;

  /* DMA0_CONFIG */
  R0.L = DI_EN | WNR;
  P0.L = lo(DMA0_CONFIG);
  P0.H = hi(DMA0_CONFIG);
  W[P0] = R0.L;

  /* DMA0_X_COUNT */
  R0.L = 256;
  P0.L = lo(DMA0_X_COUNT);
  P0.H = hi(DMA0_X_COUNT);
  W[P0] = R0.L;

  /* DMA0_X_MODIFY */
  R0.L = 0x0001;
  P0.L = lo(DMA0_X_MODIFY);
  P0.H = hi(DMA0_X_MODIFY);
  W[P0] = R0.L;
```

```
  ssync;
config_dma.END:  RTS;
```

Listing 15-2. Configure PPI Registers

```
config_ppi:

 /* PPI_CONTROL */
  P0.L = lo(PPI_CONTROL);
  P0.H = hi(PPI_CONTROL);
  R0.L = 0x0004;
  W[P0] = R0.L;
  ssync;

config_ppi.END:  RTS;
```

Listing 15-3. Enable DMA

```
  /* DMA0_CONFIG */
  P0.L = lo(DMA0_CONFIG);
  P0.H = hi(DMA0_CONFIG);
  R0.L = W[P0];
  bitset(R0,0);
  W[P0] = R0.L;
  ssync;
```

Listing 15-4. Enable PPI

```
  /* PPI_CONTROL */
  P0.L = lo(PPI_CONTROL);
  P0.H = hi(PPI_CONTROL);
  R0.L = W[P0];
  bitset(R0,0);
```

```
W[P0] = R0.L;
ssync;
```

Listing 15-5. Clear DMA Completion Interrupt

```
/* DMA0_IRQ_STATUS */
P2.L = lo(DMA0_IRQ_STATUS);
P2.H = hi(DMA0_IRQ_STATUS);
R2.L = W[P2];
BITSET(R2,0);
W[P2] = R2.L;
ssync;
```

Unique Behavior for the ADSP-BF52x Processor

None.

16 SECURITY

This chapter describes the security features and functionality of the ADSP-BF52x Blackfin processor. Following an overview and a list of key features are a description of operation and functional modes of operation.

This chapter includes the following sections:

- "Overview" on page 16-1
- "Features" on page 16-4
- "Description of Operation" on page 16-6
- "Programming Model" on page 16-32
- "Security Registers" on page 16-45

The intention of the chapter is to describe security features of the ADSP-BF52x Blackfin processor and how they can be used to facilitate a secure system. It is beyond the scope of this chapter to fully describe various ways to implement secure systems or to describe security protocols and primitives in any great detail.

Overview

Lockbox™ Secure Technology for Analog Devices Blackfin processors is comprised of a mix of hardware and software mechanisms designed to prevent unauthorized accesses and allow trusted code to execute on the

processor. Throughout the rest of this chapter, the terms Blackfin Lockbox secure technology and Lockbox will be used interchangeably.

The developer's decision to use security features is completely optional. No security features are enabled by default. The developer can choose to not implement security features in the application if it is so desired. The Blackfin will always power up/boot in Open Mode when no security features or restrictions are enabled.

Blackfin Lockbox secure technology allows users to:

- safeguard as little as a single function, a complete system, or anything in-between.
- uniquely identify each processor by a Unique Chip ID.
- utilize secure key storage provided by non-volatile, write-protectable One Time Programmable (OTP) memory.
- perform digital signature authentication using elliptic curve cryptography (ECC) and secure one-way hash (SHA-1) algorithms implemented in firmware.
- keep secret information in secure OTP Memory.
- use any encryption algorithm to protect code or other assets.
- ensure data integrity through digital signature authentication.
- safeguard confidentiality by encrypting any or all of the system from core IP (code security) to data integrity.

These features in combination provide the following benefits.

- Authenticity/Origin verification—Lockbox secure technology allows verification of a code image against its associated digital signature, and provides for a process to identify entities and data origins.
- Integrity—Developers can use a digital signature authentication process to ensure that the message or the content of the storage media has not been altered in any way. If either the message or digital signature was altered, Lockbox fails during the authentication process.
- Confidentiality—Cryptographic encryption/decryption supports situations that require the ability to prevent unauthorized users from seeing and using designated files and streams. Methods for ensuring confidentiality are supported by the secure processing environment (Secure Mode) and secure memory.
- Renewability—System components can be updated to enhance security.

  The Unique Chip ID enables end users to identify each Blackfin processor and hence each OEM device in which the processor resides.

  This Lockbox feature can be used in support of revocation and renewability of licenses in case of security violations in digital rights management systems. For example:

- Unique Chip ID—In combination with a trusted DRM agent (sourced by the OEM), this feature enables developers to implement renewability in DRM systems.
- Unique Chip ID—Provides capability to identify each OEM device and "blacklist" devices to remove them from a system.
- Prevention of mass copying—Lockbox supports cryptographic encryption/decryption algorithms for situations when confidentiality is required. The Unique Chip ID can also be utilized to "bind" the processor to one specific boot source/device and can be used to facilitate antitheft schemes and prevent OEM device cloning.

The ADSP-BF52x Blackfin processors featuring Lockbox secure technology provide security features that enable applications to use secure protocols consisting of code authentication and execution of code within a secure environment. Together these features protect secure memory spaces and restrict control of security features to authenticated developer code.

Features

Lockbox is comprised of a combination of hardware and software elements. These elements are:

- **OTP Memory**

  An array of non-volatile write-protectable memory that can be programmed by the developer only one time. Half of the array is public (accessible in any mode) and the other half is private (only accessible in Secure Mode). For more information on OTP memory, refer to Chapter 4, "One-Time Programmable Memory".

- **Secured System Switches**

Programmable bitfields in the Secured System Switches MMR to disable and enable different methods of memory access in support of a secured environment. Some of these protection mechanisms include disabling DMA access to L1 memory and disabling ADI JTAG instructions from the ICE port.

- **Secure Mode Control**

  This involves the Secure State Machine hardware required to support a transition from an unsecured state of operation (Open Mode), through an authentication state (Secure Entry Mode), and finally to a secured state (Secure Mode) where secrets are accessible.

- **Firmware**

  Code that resides in the on-chip ROM and performs digital signature authentication. Having the code that performs the digital signature authentication in ROM ensures integrity of the code.

- **User callable cryptographic ciphers**

  In addition to the control code that resides in the on-chip ROM used for authentication, the SHA-1 cryptographic function is user-callable. The API is documented in "Programming Model" on page 16-32.

- **Unique Chip ID**

  Each ADSP-BF52x Blackfin processor has a 128-bit unique chip identification value stored in public OTP memory. The Unique Chip ID is programmed and write protected before a processor leaves the Analog Devices factory. It is always located at the same OTP page address.

The 128-bit Unique Chip ID value can be read but cannot be modified by the developer or end user. A total of 64K bits of OTP memory is available to the developer if additional user-defined ID values are desired. These IDs can be stored in either public or pri-

vate areas of OTP memory depending on application requirements. Refer to Chapter 4, "One-Time Programmable Memory" for details.

Description of Operation

Blackfin Lockbox technology is based upon the concept of authentication of digital signatures using standards-based algorithms and provides a secure processing environment in which to execute code and access protected assets.

Digital signatures are created using a public-key signature algorithm, the Elliptic Curve Cryptography (ECC) public-key cipher, and a secure one-way hash algorithm, SHA-1. A public-key algorithm actually uses two different keys; the public key and the private key (called a key pair). The private key is known only to its owner and is not stored on-chip, while the public key can be available to anyone and is stored in the public OTP memory region on-chip. Public-key algorithms, such as ECC, are designed so that if one key is used for encryption, the other is necessary for decryption. Furthermore, the encryption key cannot be reasonably calculated from the decryption key. In a digital signature authentication scheme like Lockbox, the private key is used to generate the signature and the corresponding public key is used to validate the signature. Each ADSP-BF52x Blackfin processor has an on-chip ROM that contains firmware with the Elliptic Curve Cryptography (ECC) and SHA-1 algorithms. These are called to verify the digital signatures (ECDSA[1]).

JTAG emulation and test features are disabled in hardware, and certain memory access restrictions are enabled during verification of the digital signature. Once the signature is authenticated, the access restrictions are still in effect and can only be controlled by the authenticated user code.

[1] ECDSA implementation on the ADSP-BF52x Blackfin products only supports the Koblitz curve.

Secure State Machine

The ADSP-BF52x processor includes a Secure State Machine to handle the different protection configurations of the processor depending on the security situation. The machine states are “Open Mode”, “Secure Entry Mode”, and “Secure Mode” (See Figure 16-1). The following sections describe these machine states.

The state of the Secure State Machine can be identified by reading bits in the SECURE_STATUS[1:0] register. The bit values in the upper right of the states shown in Figure 16-1 correspond to the bit values in SECURE_STATUS[1:0].

For more information on the SECURE_STATUS register, see “Security Registers” on page 16-45.

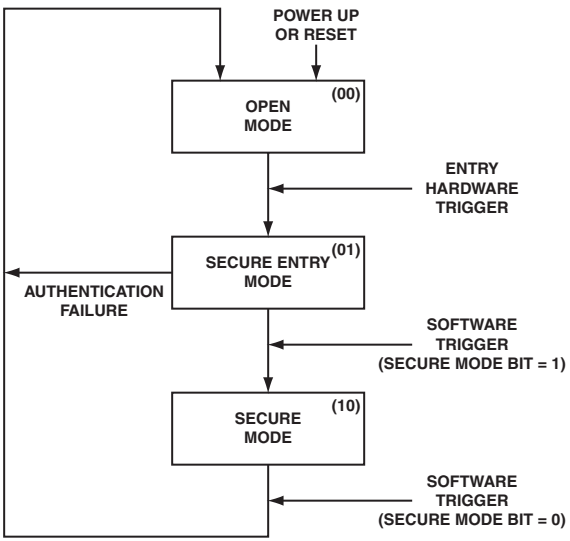


Figure 16-1. Secure State Machine Modes

Open Mode

This is the default operating state of the processor, in which no restrictions are present except restricted access to the Private OTP memory area. The processor powers up and boots in Open Mode. This is the default state upon power up and after processor reset. No Lockbox security features or protection mechanisms are enabled in this state.

The state flow illustrated in Figure 16-1 shows that the Secure State Machine can only transition from Open Mode into Secure Entry Mode, and there is no direct path from Open Mode into Secure Mode.

Secure Entry Mode

The on-chip ROM firmware performs the authentication process in this operating state. This mode is entered when NMI is active, and the program counter (PC) is vectored to the first address of the authentication firmware in the on-chip ROM. The program counter is monitored to ensure that it remains within the address range allocated to the Authentication firmware code. If the program counter vectors outside of the address range of the authorization code, authentication fails and the state returns to Open Mode. Any errors caught by firmware or hardware monitor will result in authentication failure and an abortion of the authentication process with the firmware exiting Secure Entry Mode and transitioning back to Open Mode. If authentication is successful, the firmware initiates the transition from Secure Entry Mode to Secure Mode.

In Secure Entry Mode, no DMA access is allowed to certain regions of internal SRAM, and JTAG emulation is disabled. The user should disable cache prior to initiating authentication. Interrupts are disabled by firmware prior to entry into Secure Mode. Interrupts are either re-enabled by dropping the interrupt level from NMI via the SESR arguments, or they are reenabled after authentication in the authenticated code after entry into Secure Mode. In addition, only the public area of OTP memory is accessible in this mode. For more information on memory access restric-

tions within Secure Entry Mode, see "Secure Entry Service Routine (SESR) API" on page 16-32.

State flow, illustrated in Figure 16-1, shows that the Secure State Machine can only transition from Secure Entry Mode to Secure Mode upon successful digital signature authentication. A transition from Secure Entry Mode back into Open Mode can occur if digital signature authentication fails or if the authentication process is aborted due to an error observed by the firmware. Such errors include illegal memory boundary conditions or jumps outside of the firmware range (for example, servicing an interrupt).

Secure Mode

Secure operating state in which trusted, authenticated code is allowed unrestricted access to the processor resources, execution of authenticated code occurs, decryption of sensitive information, etc. This is the only mode that allows access (reads and writes) to the private OTP memory space where secure data, such as secret keys, can be stored. Hence, the private area of OTP memory can be used to store confidential, secret information that only authorized authenticated code can access. Therefore, this is the only operating state in which users can securely run their own Blackfin implementation of any cryptographic cipher in which secret keys are used.

Only the code (or message) digitally signed by a trusted source and successfully passed through Lockbox's authentication process can gain access to Secure Mode.

State flow illustrated in Figure 16-1 shows that the Secure State Machine can only transition from Secure Mode back into Open Mode, and there is no direct path from Secure Mode into Secure Entry Mode. Exit from

Secure Mode is implemented through software control by writing a “0” value to the SECUREO bit within the SECURE_CONTROL register.

Assertion of reset or power cycling will also return the processor to the default Open Mode regardless of the state of operation when the reset or power cycle event occurred. See special handling of hardware reset in “Reset Handling in Secure Mode” on page 16-21.

Access to private OTP memory is restricted in Open Mode and Secure Entry Mode regardless of whether or not other security features are enabled or disabled.

SecureMode Control

Figure 16-2 describes the inputs that control the secure state machine flow.

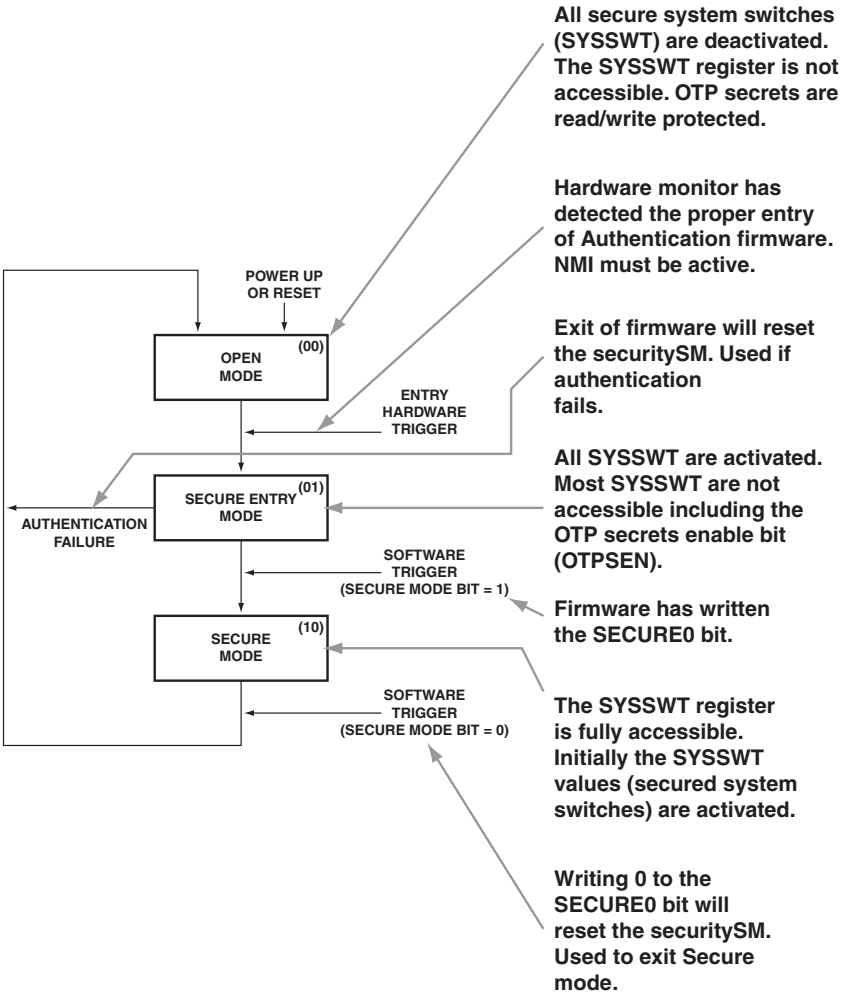

All secure system switches (SYSSWT) are deactivated. The SYSSWT register is not accessible. OTP secrets are read/write protected.
Hardware monitor has detected the proper entry of Authentication firmware. NMI must be active.
Exit of firmware will reset the securitySM. Used if authentication fails.
All SYSSWT are activated. Most SYSSWT are not accessible including the OTP secrets enable bit (OTPSEN).
Firmware has written the SECURE0 bit.
The SYSSWT register is fully accessible. Initially the SYSSWT values (secured system switches) are activated.
Writing 0 to the SECURE0 bit will reset the securitySM. Used to exit Secure mode.
POWER UP OR RESET
OPEN MODE
(00)
ENTRY HARDWARE TRIGGER
SECURE ENTRY MODE
(01)
AUTHENTICATION FAILURE
SOFTWARE TRIGGER (SECURE MODE BIT = 1)
SECURE MODE
(10)
SOFTWARE TRIGGER (SECURE MODE BIT = 0)

Figure 16-2. Secure Mode Control

Hardware supports transition from an Open Mode of operation, through a Secure Entry Mode, to a Secure Mode where secrets are accessible.

Open Mode is characterized by being the default mode of processor upon power up/reset/boot, holding all secured system switches deactivated and protecting the private OTP memory area from access. The processor is open with all features being available with no restrictions (except for the private area of OTP memory).

Secure Entry Mode is characterized by executing firmware out of internal ROM memory to authenticate information loaded into on-chip memory. All secured system switches are activated. However, private OTP Memory is not accessible yet.

Secure Mode is entered only after a successful digital signature authentication process from Secure Entry Mode. It provides access to the private OTP memory area and makes secured system switches accessible to user (authenticated) code. This is the mode of operation in which to perform sensitive decryption or execution of trusted, authenticated code.

Authentication can only be requested and initiated while the processor is operating in Open Mode. If authentication is requested while the processor is operating in Secure Mode, the Secure State Machine will not transition into Secure Entry Mode. Instead, the Secure State Machine will remain in Secure Mode.

Please note that Open Mode, Secure Entry Mode and Secure Mode are states which pertain to the Secure State Machine. User Mode and Supervisor Mode are modes of operation which pertain to the core. The use of the term “mode” should not be confused and are not necessarily mutually exclusive. In Open Mode, the processor can operate in either User or Supervisor Mode. Since the firmware is entered when the NMI is being handled, Secure Entry Mode must start in Supervisor Mode. Finally, authenticated code executing in Secure Mode must be either operating at NMI interrupt level or the interrupt level that triggered the NMI.

Security Features

The following sections provide a functional description of the Security features.

Protection relies on the on-chip ROM code that includes Elliptic Curve Cryptography (ECC) and SHA-1 algorithms, applied towards verification of code authenticity using a digital signature. A processor has emulation and test features disabled in hardware as well as certain memory access restrictions upon entry into Secure Entry Mode (where authentication is performed) and maintained into Secure Mode. These functions can be controlled only by authenticated user application software executing in Secure Mode.

User code must request authentication by complying with two criteria: (1) asserting a Non-Maskable Interrupt (NMI) and (2) vector the Program Counter (PC) to the first executable address in the Secure Entry Service Routine (SESR) in firmware which resides in on-chip boot ROM.

During the authentication process, JTAG emulation is disabled, memory protection restrictions are enabled and interrupts are masked. The user has the option to pass arguments to the security firmware to control certain functionality during the authentication process. Refer to "Secure Entry Service Routine (SESR) API" on page 16-32.

Digital Signature Authentication

Digital signatures are created off-chip (typically on a host computer) using the ECC algorithm and SHA-1, both of which are available in the public domain. In digital signature authentication, the private key generates the signature (off-chip), and the corresponding public key validates the signature (on-chip). The private key is known only to its owner and is not stored on-chip, while the public key can be available to anyone and is stored on-chip in OTP memory.

Lockbox uses standards-based cryptographic algorithms for digital signature authentication. ECDSA[1] is implemented in the Blackfin ADSP-BF52x processors. Digital signature validation on ADSP-BF52x utilizes Elliptic Curve Cryptography[2] (ECC) based on a binary field size of 163 bits and SHA-1[3] secure one-way hash (which produces a 160-bit message digest).

In order to generate public/private key pairs or prepare digital signatures and apply them to application code, developers can use any method that complies with the Elliptic Curve Digital Signature Algorithm (ECDSA) specified in FIPS 186-2 with Change Notice 1 dated October 5, 2001, Digital Signature Standard (DSS). ECDSA is described in ANSI X9.62-1998. The Lockbox implementation in the ADSP-BF52x processors supports the following Koblitz curve, which is recommended in FIPS 186-2 for US Federal Government use:

m: 163 (degree of binary field)

a: 1

b: 1 (a and b are the constants in the elliptic curve equation: $y^2 + xy = x^3 + ax + b$)

X_g: 2FE13C0537BBC11ACAA07D793DE4E6D5E5C94EEE8

Y_g: 289070FB05D38FF58321F2E800536D538CCDAA3D9 (X_g and Y_g define the base point G)

r: 4000000000000000000020108A2E0CC0D99F8A5EF (r is the order of the base point G)

T: 4 (T is the normal basis type)

p(t): $t^{163} + t^7 + t^6 + t^3 + 1$ (pt(t) is the field polynomial)

The following steps summarize the Digital Signature Authentication process. Steps 1 to 3 correspond to the off-chip creation of a digital signature

---

[1] ECDSA implementation on these Blackfin products only supports the Koblitz curve.

[2] These implementations are based on the Elliptic Curve Digital Signature Algorithm (ECDSA) specified in FIPS 186-2 with Change Notice 1 dated October 5, 2001, Digital Signature Standard (DSS) (http://csrc.nist.gov/cryptval/dss.htm), and specified in ANSI X9.62-1998.

[3] SHA-1 is based on the publicly available standard for FIPS 180-2 (Secure Hash Signature Standard [SHS]) (FIPS PUB 180-2), http://csrc.nist.gov/CryptoToolkit/tkhash.html).

of a file or message. Steps 4 to 6 correspond to the on-chip digital signature authentication. These steps are preceded by generation of a key pair (Private Key and Public Key) and the programming of the Public Key in the Public OTP Memory.

1. A one-way hash of the file (message to be authenticated) is produced using SHA-1 off-chip (for example, using a host PC).
2. The hash is encrypted through ECC off-chip with the private key, thereby signing the file and completing the generation of the digital signature.
3. The file and the signed hash are stored on an external device such as Flash memory or a host device.
4. Upon transfer to the Blackfin processor's internal memory, a one-way hash of the file is calculated on-chip through SHA-1 (residing in the Blackfin on-chip boot ROM).
5. Using the ECC algorithm (residing in the Blackfin on-chip boot ROM), the Blackfin decrypts the signed hash with the user's public key stored in the Blackfin processor OTP memory.
6. The two hash results are then compared. If the signed hash matches the calculated hash, the signature is valid and the file is intact.

If the digital signature authentication process is successful, the Blackfin processor transitions from Secure Entry Mode to Secure Mode. At this time, all of the access restrictions mentioned will be in place. JTAG will be disabled and certain portions of on-chip SRAM memory are restricted from DMA access. The restrictions can be controlled once in Secure Mode by having the authenticated code modify the Secure System Switches (`SECURE_SYSSWT`) appropriate for use by the developer's application.

Encryption/decryption is only necessary when an application requires *confidentiality*. It is not always necessary to work with encrypted code to ensure code security. Authentication alone can

be used when confidentiality is not required when ensuring tamper-proof code image and/or non-repudiation in a system. Thus, authentication safeguards code *integrity*.

Since the digital signature uniquely describes its corresponding code/message, the code/message itself does not have to be encrypted if *confidentiality* is not required. If the code/message is modified, either intentionally or inadvertently, authentication fails since the *integrity* of the code message has been compromised.

Digital Signature Authentication Performance Measurement

Authentication can be performed at any point during processor operation in Open Mode. It can be performed immediately upon boot or it can be performed any time after boot.

The algorithms used in the Lockbox firmware are highly optimized Blackfin code running from the on-chip boot ROM in the system clock domain. Firmware execution time for the digital signature authentication process is on the order of 40 million core clock cycles, depending upon the size of the digitally signed application code. This must be considered when architecting an application in order to allow a sufficient window of time in which authentication can proceed without requiring servicing of interrupts in the system.

The time it takes for authentication is dependent on several factors. These include the size of the message to be authenticated. This affects the amount of calculations done in the secure hash function (SHA-1). It also affects the DMA time required to move the message out of L1 data memory and place it into L1 code memory.

Protection Features

In order to establish a secure processing environment and protect the security of applications that establish trust and reach the privileged mode of

operation, Lockbox implements access restrictions. These restrictions include disabling JTAG emulation and disabling DMA access to portions of on-chip SRAM memory. The memory access restrictions implemented in hardware on the Blackfin processor are not applied to off-chip memory. Therefore, external memory is always considered insecure and caching external memory while operating in Secure Mode represents a security risk.

Protection features include the following:

- Secure State Machine for implementing privileged states of operation in which access restrictions may be imposed to protect code and data.
- Disable DMA access to L1 memory
- These restrictions to memory areas are configurable (see "Secure System Switch Register, Bits 15:0" on page 16-47)
- Protection of L1 regions of memory with DMA access controlled when in Secure Mode
- Disable ADI JTAG emulation from ICE port
- Divert hardware reset to NMI during Secure Mode operation to prevent "reset attack"
- Provide software control over hardware protection features accessible to trusted code operating in Secure Mode
- OTP memory for storage of customer programmable cipher keys, unique chip ID or a customer ID
- OTP write protection to protect programmed OTP memory locations from future tampering
- Private/Secret OTP memory region accessible only in Secure Mode

- Store private key(s) for decryption of data or other validation
- A privileged mode (including firmware execution out of on-chip ROM) to perform code authentication

Protection mechanisms are summarized Table 16-1 for each state of the Secure State Machine along with the Secure System Switch register (SECURE_SYSSWT) that provides control over the protection feature.

Table 16-1. Secure State Machine

Secure State Machine	SECURE_SYSSWT	Description	Protected Memory Range
Open Mode (0x00000000)	The switches are involuntarily set with all controls OFF (unrestricted access)	No protection mechanisms or restrictions enabled	No restrictions[1]
Secure Entry (0x000704D9)	EMUDABL	Emulation Disable	Emulation disabled
	L1IDABLE	L1 Instruction Memory Disable 0xFFA00000—0xFFA07FFF SRAM	32 KB
	L1DADABL	L1 Data Bank A Memory Disable 0xFF800000—0xFF807FFF SRAM and SRAM/Cache	32 KB
	L1DBDABL	L1 Data Bank B Memory Disable 0xFF900000—0xFF901FFF SRAM	8 KB

Table 16-1. Secure State Machine (Continued)

Secure State Machine	SECURE_SYSSWT	Description	Protected Memory Range
Secure Mode (0x000704D9)	EMUDABL	Emulation Disable	User Config-urable
	RSTDABL	RESET Disable	User Config-urable
	L1IDABLE	L1 Instruction Memory Disable 0xFFA00000—0xFFA07FFF SRAM	0-32 KB
	L1DADABL	L1 Data Bank A Memory Disable 0xFF800000—0xFF807FFF SRAM and SRAM/Cache	0-32 KB
	L1DBDABL	L1 Data Bank B Memory Disable 0xFF900000—0xFF901FFF SRAM	0-32 KB

1 Private OTP is only accessible when operating in Secure Mode with OTPSEN bit set in SECURE_SYSSWT register

On-chip SRAM memory protection takes the form of DMA access restrictions only. There is no need to protect the on-chip SRAM from processor core access because, while operating in Secure Mode, the developer's authenticated code has full control over the processor core and execution of all core software instructions. It is the responsibility of the developer to take steps to avoid surrendering control of the Program Sequencer and the core to untrusted code execution.

Operating in Secure Mode

Entering Secure Mode

Upon successful digital signature authentication, the Secure State Machine transitions into Secure Mode. The same default protection fea-

tures enabled in Secure Entry Mode are carried forward into Secure Mode. This includes JTAG emulation being disabled, and DMA access restrictions to memory and interrupts being masked. It is the responsibility of the authenticated code to manipulate or remove these restrictions as desired.

Exiting Secure Mode

Secure Mode provides a secure operating environment to execute sensitive code, run cryptographic ciphers, and process sensitive data. Upon exiting Secure Mode, the authenticated code should remove any sensitive code and data from memory because this sensitive information will still be accessible in Open Mode if it is not removed prior to exiting Secure Mode. Exit from Secure Mode is implemented through software control by writing a "0" value to the SECURE0 bit within the SECURE_CONTROL register. Refer to "Security Registers" on page 16-45 and "Clearing Private Data" on page 16-22 for more information.

Reset Handling in Secure Mode

Hardware Reset

Hardware reset is diverted to NMI when operating in Secure Mode only. When operating outside of Secure Mode, hardware reset behaves normally. This protection feature is configurable via the RSTDABL bit within the SECURE_SYSSWT register when operating within Secure Mode.

This is a protection feature to prevent malicious entities from attempting to assert hardware reset while sensitive code or data is present in the processor's on-chip SRAM or in the processor's registers. A "reset attack" could take the following form: If hardware reset were left unprotected and reset was asserted while sensitive information were present on-chip, the processor would return to the default state of Open Mode with no protection features enabled and a malicious entity could gain access to the

on-chip memory and registers, for example via JTAG emulation. In such a scenario assets intended to be protected could be compromised.

By diverting hardware reset to NMI while the processor operates in Secure Mode, servicing of hardware reset can be controlled and delayed in order to first implement a memory clean-up routine in software to purge sensitive information from internal memory and registers prior to servicing reset. At the completion of the memory clean-up, the processor can then be reset via software command and safely returned to Open Mode with no sensitive information available to be compromised.

By default, the SESR loads the address of a memory clean-up routine stored in the on-chip boot ROM into the NMI `EVT2` prior to transitioning from Secure Entry Mode into Secure Mode. See "Clearing Private Data" on page 16-22 for more information.

Clearing Private Data

As part of the SESR firmware, there is a small routine stored in the on-chip boot ROM that clears the internal L1 data memory, generates a `RESET` event, and puts the processor into idle. Note that this firmware memory clear routine does not clear the contents of L1 Instruction memory or Data, Pointer, and DAG registers within the computational units. It is recommended that the user sets this routine as the new `EVT2` NMI vector once the user's authenticated application code is executing. This will prevent a malicious user from trying to reset the processor while it is operating in Secure Mode and then view the contents of internal memory when the processor returns to Open Mode after servicing `RESET`.

It is recommended that user software running in Secure Mode should also perform RAM clean-up prior to clearing the `SECURE0` Secure Mode bit and exiting Secure Mode via normal code execution within user's secure function. If sensitive code/data remains in on-chip RAM after exiting Secure Mode without wiping memory and register contents or cycling power to the processor, it is visible and accessible in Open Mode.

The memory clear routine in the on-chip boot ROM executes a watchdog RESET to reset the processor at the completion of the memory clear. The code also performs a clear of the OTP_DATA0-3 registers which are used to hold data from OTP access reads (i.e. which could contain secret key or other sensitive data left by user code execution).

If a custom memory cleanup routine is part of an authenticated message, the user can use that routine instead of the one provided with the Lockbox firmware. The user can simply update EVT2 in the event vector table to point to the start of the custom memory cleanup routine while operating in Secure Mode.

It is strongly recommended that developers substitute their own custom memory clear routines if they require clearing of L1 instruction as the ROM memory clear routine will only clear the contents of L1 Data (Bank A and B) memory. The ROM memory clear routine will not protect instruction code from being exposed after reset is serviced or when the Secure State Machine transitions to Open Mode via other means.

Due to the fact that hardware reset is configured by default to be redirected to NMI when the processor is operating in Secure Mode, it is recommended that the user implements a watchdog reset within the EVT2 NMI ISR in order to reset the processor. A Watchdog reset is implemented by writing a value 2'b00 in WDOG_CTL[2:1] and causes a complete core reset. The watchdog reset will not be redirected to the NMI pin as in the case of the external hardware reset and it will properly reset the processor. For more details of watchdog reset, refer to "Software Resets" in Chapter 17, System Reset and Booting.

This "reset attack" protection scheme needs to protect only against hardware reset. Since it can be applied externally, the system developer typically has no control over reset in an embedded system. While operating in Secure Mode, the developer's authenticated code has full control over the processor core and execution of all software instructions, so there is no need to protect against soft reset instructions. It is not recommended

that the user's secure application code implement a soft reset without first deleting sensitive information from memory and registers.

Public Key Requirements

A valid ECC public key must be a non-zero value and meet the following criteria:

Given the public key value shown here:

369368AF243193D001E39CE76BB1D5DA08A9BC0A6

15F7A90C841D4F1E1B005E70F167F6EF7CD2E251B

format in 32-bit little endian as follows:

8A9BC0A6

BB1D5DA0

1E39CE76

43193D00

69368AF2

00000003

CD2E251B

167F6EF7

B005E70F

41D4F1E1

5F7A90C8

00000001

The values should be stored in OTP pages 0x10, 0x11, 0x12 as follows, where 'L' denotes lower half of page (OTP page bits 63:0), 'H' denotes upper or high half of page (OTP page bits 127:64):

page: 0x010L: 0xbb1d5da08a9bc0a6,

page: 0x010H: 0x43193d001e39ce76,

page: 0x011L: 0x0000000369368af2,

page: 0x011H: 0x167f6ef7cd2e251b,

page: 0x012L: 0x41d4f1e1b005e70f,

page: 0x012H: 0x000000015f7a90c8,

The general format takes the form of twelve (12) 32-bit words:

Word 1

Word 2

Word 3

Word 4

Word 5

Word 6

Word 7

Word 8

Word 9

Word 10

Word 11

Word 12

Stored into OTP pages in the following order (where 'L' denotes lower half of page, 'H' denotes upper or high half of page):

page: 0x010L:Word 2 Word 1

page: 0x010H:Word 4 Word 3

page: 0x011L:Word 6 Word 5

page: 0x011H:Word 8 Word 7

page: 0x012L:Word 10 Word 9

page: 0x012H:Word 12 Word 11

Storing Public Cipher Key in Public OTP

In order to make use of security features, the user must first store an ECC public key in the Blackfin processor public region of OTP memory pages 0x10, 0x11, and 0x12 as specified in the Firmware's Secure Entry Service Routine (SESR) API and the OTP memory map (see "Secure Entry Service Routine (SESR) API" on page 16-32). If no ECC public key is stored in this area of OTP, digital signature authentication cannot be successfully

completed and no Lockbox security features can be enabled. For more information see Chapter 4, “One-Time Programmable Memory”.

Cryptographic Ciphers

Lockbox uses SHA-1 and ECC to implement ECDSA as part of the authentication process to enter into Secure Mode. These ciphers reside in the firmware in the on-chip boot ROM. The SHA-1 cipher is user-callable in Open Mode or in Secure Mode. The API is documented in “Programming Model” on page 16-32. Note that ECC is not user-callable and is only executed as part of firmware during the authentication process.

Keys

Although Lockbox uses an ECC public key for digital signature authentication and has private OTP memory to store private keys for other cryptographic algorithms, Lockbox does not implement key management. Lockbox does not implement key generation, nor does it implement key exchanges natively in the Blackfin hardware.

In order to use Lockbox, an ECDSA key pair must be generated. The private key is used off-chip (typically on a host PC) to sign the message. The public key is placed in the public OTP memory where it is used to authenticate the signed message. Lockbox is only part of a full cryptosystem. It is the responsibility of the user to develop the other parts of the cryptosystem necessary for the intended application.

Debug Functionality

The processor is fully compatible with the IEEE 1149.1 standard, also known as the Joint Test Action Group (JTAG) standard. Full details of the JTAG standard can be found in the document IEEE Standard Test Access Port and Boundary-Scan Architecture, ISBN 1-55937-350-4.

ADSP-BF52x debug functionality has some modified behavior dependent upon the access privileges associated with the state of the Secure State Machine operating mode. This is to ensure that sensitive information and processing performed within Secure Entry Mode and Secure Mode will not be compromised via JTAG. Furthermore, public JTAG instructions necessary for system test and debug (such as boundary scan and bypass mode) remain in effect regardless of the state of the Secure State Machine and are not hindered by the ADSP-BF52x Secure Mode operation. This makes it possible for developers to debug their systems without interference from the Blackfin processor or its security features.

In compliance with the JTAG standard, ADSP-BF52x processors provide an Instruction Register (IR) that interprets 5-bit instruction codes to select the test mode that performs the desired test operation. The instruction register is five bits wide and accommodates up to 32 boundary-scan instructions. The instruction register holds both public and private instructions. The JTAG standard requires some of the public instructions; other public instructions are optional. Private instructions are reserved for the manufacturer's use.

All supported public and private JTAG instructions remain operational when operating in Open Mode. All supported public JTAG features remain operational and all private JTAG features are disabled when operating in Secure Entry Mode and Secure Mode. Refer to Appendix B, "Test Features" for more information about supported JTAG instructions

By default, JTAG emulation is disabled when the processor enters Secure Entry Mode or Secure Mode. There is only one way to enter Secure Mode—through successful authentication of user code based on digital signature validation. Once the digital signature authentication process results in success, the user's trusted, authenticated code is given full control over the processor, including access to Secured System Switches register (`SECURE_SYSSWT`) that enables/disables various protection mechanisms, including JTAG emulation. The Secured System Switch register provides a setting that allows authenticated code to enable JTAG emula-

tion either in a one-time secure session setting or in a “sticky” persistent manner that allows emulation to be enabled by default the next time the processor enters Secure Mode. These settings are cleared when reset is asserted or if processor core power is cycled. See the EMUOVR and EMUDABL bits in the SECURE_SYSSWT Secure System Switches Register in “Secure System Switch Register, Bits 15:0” on page 16-47.

Two bits within the SECURE_SYSSWT Secure System Switches register control JTAG emulation; they are Emulation Disable (EMUDABL) and Emulation Override (EMUOVR). To enable JTAG emulation for the current session while operating within Secure Mode, SECURE_SYSSWT bit 0 (EMUDABL) must be set to 0. To enable JTAG emulation to remain “sticky” and persistently enabled for the current session and for all subsequent entries into Secure Mode until cleared by the user or until cleared via RESET or cycling power to the processor, SECURE_SYSSWT bit 0 (EMUDABL) must be set to 0 *and* SECURE_SYSSWT bit 14 (EMUOVR) must be set to 1 simultaneously. See “Secure System Switch Register, Bits 15:0” on page 16-47 for details.

The EMUDABL bit is only directly writable when in Secure Mode. EMUOVR can be written to a 0 at any time. RESET will clear EMUOVR. EMUOVR can be cleared by the user at any time and in any mode, including Open Mode, Secure Entry Mode, and Secure Mode. You do not have to operate in Secure Mode in order to clear EMUOVR.

The EMUDABL bit is writable only directly when in Secure Mode. EMUOVR can be written to a 0 at any time. This means if you are in Secure Mode and wish to remove the privilege of emulation override, you are allowed to clear EMUOVR. Or if you are operating in Open Mode and wish to remove emulation override, you can clear EMUOVR. In the case of Secure Entry Mode, writing the EMUOVR bit to a 0 immediately blocks emulation (and the EMUDABL bit would read 0 immediately). While Operating in Secure Entry Mode, the value of EMUDABL is the *not* of EMUOVR, i.e., EMUDABL = ~EMUOVR. While operating in Secure Mode, you can read or write the

EMUOVR bit, which has no immediate affect since EMUDABL is in control at that point.

Upon setting EMUDABL = 0 *and* EMUOVR = 1, JTAG emulation remains active and enabled for the current session during Secure Mode operation *and* for *all* subsequent entries into Secure Mode until EMUOVR is cleared (set to 0) or until RESET or power cycle clears this setting. This is also known as "sticky" emulation setting.

If "sticky" emulation is enabled (EMUDABL = 0 *and* EMUOVR = 1), JTAG emulation is active and enabled in all modes, that is, Secure Entry, Secure Mode, as well as Open Mode. The Secure State Machine can cycle through all modes of operation, and JTAG emulation will remain active and enabled in every mode with these settings in place until cleared by the user application code, or until RESET or power cycle clears the setting.

For example, a user creates code to be authenticated with a valid digital signature. The code and digital signature are loaded onto the Blackfin processor in Open Mode, Authentication is requested (JTAG emulation is disabled by default during Authentication in Secure Entry Mode), and the Authentication process is successful. The processor enters Secure Mode (JTAG emulation still is disabled by default) and control is given to the authenticated code. Authenticated code sets bits within the Secure System Switches to enable JTAG Emulation and sets the "sticky" bit to allow JTAG emulation to be enabled by default the next time the processor transitions into Secure Mode as well. Debug within Secure Mode can occur using emulation now. If a different set of trusted code must be loaded into the processor, the user can do so now without leaving Secure Mode, or the user can choose to exit Secure Mode and return back to Open Mode in order to authenticate another set of code or load test/problematic code. A new set of code and digital signature now can be loaded and authenticated. Upon entry into Secure Mode, JTAG emulation will be enabled by default due to the sticky bit setting in the Secure System Switches. Debug can be performed within Secure Mode without changes to problematic code.

One possible usage scenario for persistent (sticky) emulation might be as follows: a “final” production code that must run in Secure Mode is prepared. There seems to be an issue with the code, but emulation prevents working with it. You would take advantage of the EMUOVR bit within the SECURE_SYSSWT register by first performing a simple authentication of code that sets the EMUOVR bit in order to enable JTAG emulation within Secure Mode. From there you exit Secure Mode (write a value of “1” to the SECUREO bit in the SECURE_CONTROL register, but do not invoke any processor reset), and call the routine to debug. You would then set a breakpoint just after authentication. That way you can now step through your code using JTAG emulation and operate in Secure Mode.

Digitally signed user code, which enables either single session or sticky JTAG emulation, must be treated as confidential by users in the same manner as private keys. If this code is allowed to fall outside of developer control or become public, it can be used to compromise a developer’s security.

In summation, to enable JTAG emulation during Secure Mode, the user must successfully perform the Authentication process at least one time, and then program the Secured System Switches while operating in Secure Mode to enable emulation.

Programming Examples

Listing 16-1. Assembly Code – Enable (“Sticky”) Persistent JTAG Emulation for Secure Mode Debug

```
#include <defBF527.h>
.section L1_code;
.align 4;
.global _secure_function;
_secure_function:
/* required nops to account for SESR PC vector target+4 for
overlay ID accommodation*/
```

```
  nop;
  nop;
  P0.H = ((SECURE_SYSSWT) >> 16);
  P0.L = ((SECURE_SYSSWT) & 0xFFFF);
  R0 = [P0];
  BITCLR(R0,0);
  [P0] = R0;
  SSYNC;
  _secure_function
.END:
```

Listing 16-2. C Code – Enable JTAG Emulation for Secure Mode Debug (single session)

```
#include <cdefBF527.h>
#define    ENABLE_JTAG_MASK       0xFFFFFFFE
void secure_function(void)
{
  /* Enable JTAG */
  *pSECURE_SYSSWT = ( *pSECURE_SYSSWT & ENABLE_JTAG_MASK );
  ssync();
  return;
}
```

Programming Model

Secure Entry Service Routine (SESR) API

This section describes the procedure to use Lockbox to authenticate a message. Memory configuration, input arguments and return codes are also described here.

In this chapter, the term “message” was widely used to describe the entity being digitally signed off-chip, and later authenticated on-chip by the SESR security firmware. “Message”, “secure function” (SF), and “secure application” are used interchangeably in this section and mean the same thing.

Starting Authentication

For an application to establish trust and reach the privileged mode of operation (for example, enter Secure Mode), the Secure State Machine has to transition from Open Mode, through Secure Entry Mode, to Secure Mode. In order to transition from Open Mode to Secure Entry Mode, NMI must be asserted and the program counter (PC) must vector to the beginning address of the firmware (SESR) at location 0xFFA1 4000.

This can be achieved by loading the beginning address of the SESR (0xFFA1 4000) as the NMI handler in the event vector table (`EVT2`). Then in supervisor mode, issue a `raise 2;` instruction. Similarly, NMI hardware pin may be asserted instead of issuing a software `raise;` instruction. Once the program counter vectors to the SESR, while NMI assertion is sensed by the hardware, the Secure State Machine transitions into Secure Entry Mode.

Before actually going into Secure Entry Mode, the user will have to set up the memory environment. This includes specifying the arguments (described in this section) and moving the message to be authenticated into L1 data memory.

Memory Configuration

Figure 16-3 illustrates the Secure Entry Mode default memory configuration upon initiating authentication and entering the SESR.

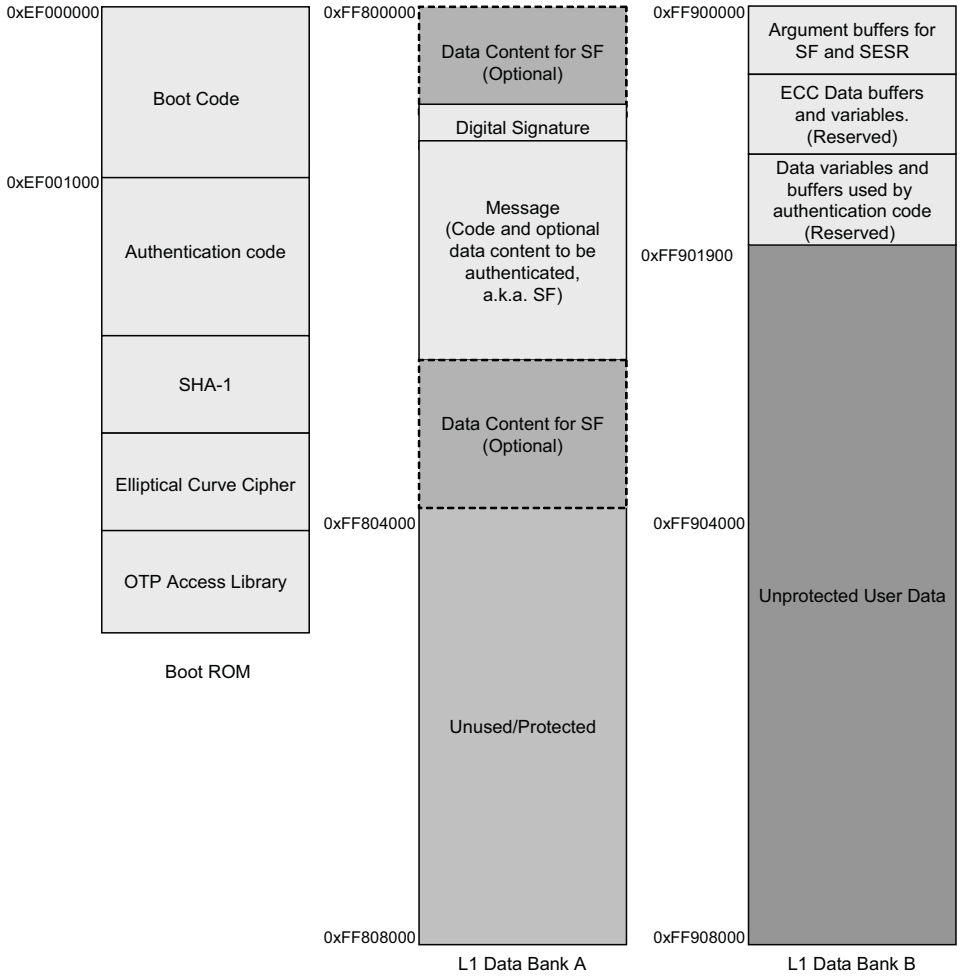


Figure 16-3. Memory Configuration for Authentication

Message Placement

The message must be placed in L1A for authentication. If the message (for example, code) is put into L1A for authentication, it must be DMA'd to L1 code space, where it can execute. It is the user's responsibility to provide the message in L1A memory for the SESR. If authentication is successful, the SESR then moves the message via DMA to the final destination according to the SESR arguments. No further action is required by the developer to perform this DMA as it is executed by the firmware.

Digital Signature

The digital signature is a pair of 163-bit integers. Each integer is padded to the nearest 32-bit word, resulting in 192 bits for each integer, resulting in a total size of 384 bits. The authentication firmware always expects the digital signature to be followed by the message. For example, if the message is placed in L1A data memory, and the digital signature starts at address 0xFF80 0000, the message must immediately follow the digital signature and be located at address 0xFF80 0030. The message and digital signature must be stored together contiguously in memory with the message always immediately following the digital signature.

Message Size Constraints

The maximum size of any message to be authenticated is limited by the size of on-chip memory in the Blackfin processor. When the Secure State Machine enters into Secure Entry Mode (authentication), certain portions of on-chip SRAM memory are protected from DMA accesses. These protected memory regions include L1A (32 KB) and L1B data memory (8 KB each and 32 KB of L1 code memory). This means that the maximum allowable message/code size that can be authenticated is 32 KB less 48 bytes for the digital signature when placed in L1A data memory.

Memory Usage

In data bank B of the L1 memory, the arguments for both the SESR and the secure function are stored beginning at address 0xFF90 0000. In addition, a portion of the L1B data memory is reserved for the firmware for scratch space. All memory above address 0xFF90 1900 is reserved for authentication. The user can either allocate this area of memory solely for Lockbox or save any data elsewhere in memory prior to starting authentication.

Any user information residing in the scratch space reserved area of L1 Data Bank B will be overwritten during the authentication process.

Memory Protection

This Secure Entry Mode default memory configuration with both protected and unprotected regions of on-chip SRAM is implemented in order to allow developers to initiate digital signature authentication at any time during Open Mode processor operation. If an application is already running on the processor, the unprotected memory regions can be used for placement of data buffers. When authentication occurs, access to these data buffers is not restricted, thus the application can be given higher precedence over the authentication process if necessary.

The Secure Entry Mode default memory protection configuration put into place upon initiating authentication cannot be modified by the developer. This is to ensure integrity of the secure processing environment during the authentication process and help prevent malicious tampering.

Secure Function and Secure Entry Service Routine Arguments

Prior to initiating the authentication, the arguments for both the SESR and the message (also known as Secure Function or SF) must be set up.

The arguments are stored in argument buffers stored in L1B data memory. Specifically, the arguments for the Secure Function are stored at the top of L1B data memory, at address 0xFF90 0000. There are 24 bytes allocated for the arguments for the secure function. Following the argument buffer for the Secure Function is the argument buffer for the SESR, at address 0xFF90 0018. For security reasons this authentication protocol accesses fixed locations for arguments. When the user starts executing the Secure Function, it receives two arguments. The first argument (`R0`) contains the address of the Secure Function argument buffer. The second argument (`R1`) holds the `IMASK` value before shut off interrupts.

Secure Function Arguments

When the message is successfully authenticated, the Program Counter will vector to the Secure Function with the first argument (`R0`) containing a pointer to top of L1B data memory. The second argument (`R1`) of the secure function is the `IMASK` value. This value is obtained when the SESR successfully authenticates the message. Before the message is transferred via DMA to its final target run location, interrupts are shut off so tampering cannot occur between the time of successful authentication and execution of the secure function. The prototype for the secure function is:

```
void secure_function(tSecureFunctionArgs *, unsigned short
imask);
```

The 24-byte Secure Function argument buffer is for the convenience of the user to be able to pass arguments to the Secure Function prior to starting authentication.

It is the responsibility of the user's Secure Function to re-enable interrupts by using the saved `IMASK` value or by using a new `IMASK` value.

The 24-byte Secure Function argument buffer can be used in any aligned fashion. For example, it can be used to store six 32-bit words or twelve 16-bit words, or any combination of data types such as integers, shorts and characters, as long as the accesses are aligned.

Secure Entry Service Routine Arguments

The argument buffer for the SESR is shown in Listing 16-3.

Listing 16-3. Argument Buffer for SESR

```
/* SESR argument structure. Expected to reside at address
0xFF900018 */
typedef struct SESR_args {

unsigned short usFlags;/* security firmware flags*/
unsigned short usIRQMask;/* interrupt mask*/
unsigned long ulMessageSize;/* message length in bytes*/
unsigned long ulSFEntryPoint;/* entry point of secure function*/
unsigned long ulMessagePtr;/* pointer to the buffer containing
the digital signature and message */
unsigned long ulReserved1;/* reserved*/
unsigned long ulReserved2;/* reserved*/
} tSESR_args;
```

usFlags

The first argument, `usFlags`, is a 16-bit flag that signals authentication what to do. Figure Figure 16-4 shows the meaning of the bits.

Bit 0 tells the authentication firmware whether or not to drop the interrupt level. To execute `raise 2;`, the Blackfin processor must operate in supervisor mode, in other words, operate at one of the interrupt levels. NMI must be asserted when authentication is initiated. The caller/user has the option to deassert NMI and drop back down to a lower interrupt

level (the interrupt level in effect when NMI was asserted to initiate authentication) or continue authentication at NMI level.

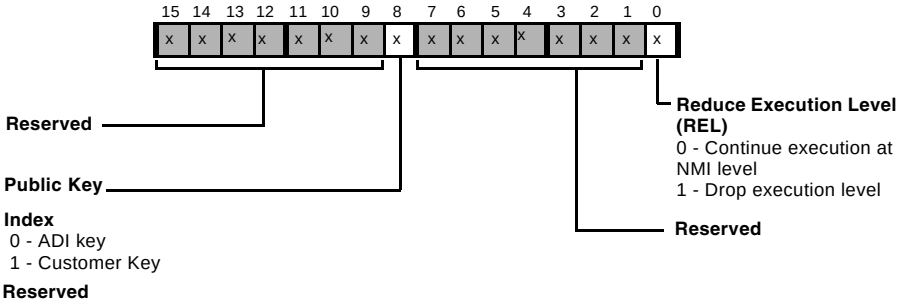

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Reserved
Public Key
Index
0 - ADI key
1 - Customer Key
Reserved
Reduce Execution Level (REL)
0 - Continue execution at NMI level
1 - Drop execution level
Reserved

Figure 16-4. Bit Fields for Flags Argument

By lowering the interrupt level at which the authentication firmware executes, other interrupts can be serviced. Be aware that if another interrupt is serviced and the PC vectors out of the authentication firmware during authentication, the authentication process fails and returns an error code.

Bit 8 tells the firmware which public key is used for authentication. The OTP memory holds two public keys. One is programmed by Analog Devices for failure analysis purposes only, and the other is programmed by the developer.

uslRQMask

The `usIRQMask` argument is a 16-bit user-defined bitmask to be loaded into the lower 16 bits of the `IMASK` MMR if the execution level is to be lowered from NMI level. This argument allows the user to specify which, if any, interrupts will be allowed to be serviced should they occur during the time authentication occurs. Note that if any interrupt is serviced, the authentication process fails and returns an error code as mentioned above.

For more information regarding `IMASK`, please refer to the Blackfin Processor Programming Reference manual.

ulMessageSize

The `ulMessageSize` argument is a 32-bit non-negative integer that tells the SESR how big the message is, in bytes. The `ulMessageSize` must be a multiple of two, otherwise the SESR returns an error code.

ulSFEntryPoint

The `ulSFEntryPoint` argument is the final address that the message will be moved to and executed from L1 Instruction memory. Again, since the authentication firmware expects code as the first portion of the message, the address must be a multiple of four since instructions can be either 16-bit or 32-bit lengths. If the message consists of both code and data, it is the user's responsibility to move the data to the proper area of data memory for subsequent use within the application.

ulMessagePtr

The `ulMessagePtr` argument holds the address where the digital signature and message are found in L1 Data memory.

Secure Message Execution

If the authentication of the digital signature is successful, the authentication firmware directly vectors the program counter to the Secure Function at its final target location, plus an offset of four bytes. The offset provides a location for the overlay ID if overlays are used with Lockbox. To return to the calling function, the authenticated message must execute `rtn;` if execution level was not signaled to be lowered in the authentication firmware. Otherwise, if the execution level was lowered, the Secure Function can return via `rts;`.

To prevent tampering, interrupts and the watchdog timer are shut off near the end of successful authentication. It is the user's responsibility to re-enable the interrupts and the watchdog timer in the Secure Function if they are required in the user's application while operating in Secure Mode.

Return Codes

If for any reason an error occurs, the SESR returns an error code, and bit 7 in the SECURE_STAT MMR sets to indicate that register R0 contains a valid error code. Table 16-2 lists a portion of the valid return codes.

Table 16-2. List of Return Codes from SESR

Return Codes	Value	Description
SECFW_SUCCESS	0	Success
SECFW_ERROR_INV_FLAGS	–1	"Flags" argument to firmware is invalid
SECFW_ERROR_INV_INTMASK	–2	IRQ mask specified is invalid
SECFW_ERROR_INV_CODESZ	–3	Code size specified is either non-positive or odd
SECFW_ERROR_OOB_CODE	–6	The message (Secure function) is too big and surpasses the protected region in L1A
SECFW_ERROR_BAD_EVT	–10	One of the ISR specified in the Event Vector table points inside the authentication firmware.
SECFW_ERROR_PUBKEY_ZERO	–11	Invalid public key of (0,0)
SECFW_ERROR_AUTH_FAILED	–12	Invalid message/signature pair
SECFW_ERROR_DMA	–15	MDMA error occurred during DMA transfer or the message to the final target vector.
SECFW_ERROR_DROPPING_INT_FAILED	–17	Could not drop interrupt level from NMI.
SECFW_ERROR_FUSE_READ_FAILED	–18	Error occurred while reading OTP memory.

Table 16-2. List of Return Codes from SESR (Continued)

Return Codes	Value	Description
SECFW_ERROR_TGTVECT_NONALIGNED	–19	Target vector is not 4 Byte aligned.
SECFW_ERROR_SECURE0_WRITE_FAILED	–20	Write to Secure0 bit failed. Secure State Machine might be blocking the write because ISR was taken.
SECFW_ERROR_SM_NOT_ENTERED	–21	Secure0 bit was written three times but secure mode was still not entered.
SECFW_ERROR_BAD_TGT_ADDR	–22	Target vector must be in L1 code space.
SECFW_ERROR_SF_TOO_BIG	–23	Message (Secure function) too big to fit at target location.

In addition to the return codes listed in Table 16-2, a return value between –62 and –252 is also a valid error return code. These errors are from OTP accesses.

To decipher the error from an OTP access, there is an offset that must be added to the error code. The macro `OTP_READ_ERROR_OFFSET` (defined in VisualDSP++ header files with a value of –285) is added to the return

value. The result is a bit mask. Figure 16-5 shows the definition of the bit fields.

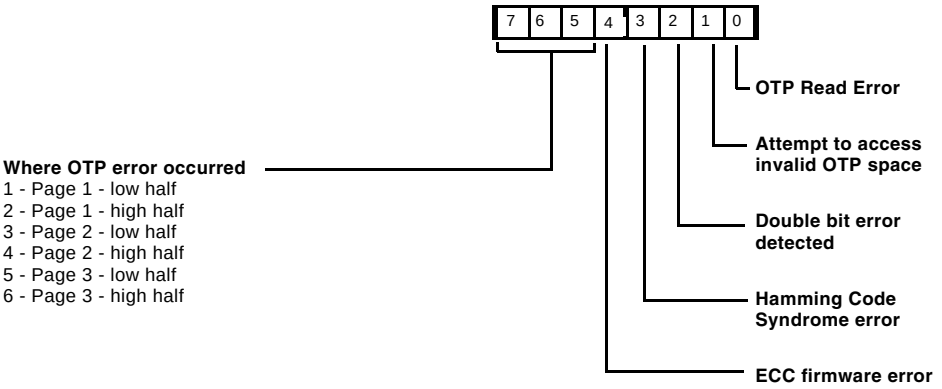


Figure 16-5. Bit Field Definition Return Value if OTP Error Occurred

Secure Hash Algorithm (SHA-1) API

The ADSP-BF52x processor includes a software implementation of the Secure Hash Algorithm (SHA-1) in the on-chip boot ROM. This implementation of the SHA-1 hash algorithm is C-callable.

The following describes the application programming interface (API) for using SHA-1, including both data types and ROM routines.

ADI_SHA1 Data Type

```
typedef struct ADI_SHA1 {
u8   *pInputMessage;
u32  udMessageSize;
u8   *pOutputDigest;
u8   *pScratchBuffer;

} ADI_SHA1;
```

The SHA1 hash routine, bfrom_Sha1Hash, when provided with a reference to an object of type ADI_SHA1, hashes the udMessageSize-long message referenced by pInputMessage, and stores the hash value (also referred to as message digest) in the buffer referenced by pOutputDigest. The elements in an object of type ADI_SHA1, are shown in Table 16-3.

Table 16-3. Elements in an Object of Type ADI_SHA1

pInputMessage	Pointer to the input buffer
udMessageSize	The size, in bytes, of the valid input data in pInputMessage.
pOutputDigest	Pinter to the output data buffer. After hashing, this buffer will contain the digest of the input message. The digest is 160-bits (SHA1_HASH_SIZE-bytes) long
pScratchBuffer	Pointer to a data buffer of size, SHA1_SCRATCH_BUFFER_SIZE-bytes, used by the SHA-1 module.

bfrom_Sha1Init ROM Routine

Entry address: Defined as BFROM_SHA1_INIT in the bfrom.h header file in the VisualDSP++ installation directory.

Arguments:

R0: Pointer to a buffer of size SHA1_SCRATCH_BUFFER_SIZE

C prototype:

```
void bfrom_Sha1Init (u8 *pScratchBuffer);
```

This function initializes some data elements in pScratchBuffer. It is called first before making any calls to bfrom_Sha1Hash.

bfrom_Sha1Hash ROM Routine

Entry address: Defined as BFROM_SHA1_HASH in the bfrom.h header file in the VisualDSP++ installation directory.

Arguments:

R0: Pointer to an object of type ADI_SHA1

C prototype:

```
void bfrom_Sha1Hash (ADI_SHA1 *pSha1);
```

This function performs the hash operation.

Security Registers

There are three registers for security mode control and status of the Secure State Machine states. These registers require privileged access depending upon the operating state of the processor.

Table 16-4. Security Registers

Register	Description	Size (Bits)	Memory-Mapped Address
SECURE_SYSSWT	Secure System Switches	32	0xFFC03620
SECURE_CONTROL	Secure Control	16	0xFFC03624
SECURE_STATUS	Secure Status	16	0xFFC03628

Secure System Switch (SECURE_SYSSWT) Register

The SECURE_SYSSWT register controls hardware that would otherwise allow a threat of attack to a secured system. Hardware is controlled voluntarily and involuntarily as follows.

- During Open Mode the switches are involuntarily set with all controls off (unrestricted access, with exception of access to OTP protected “secrets” area). OTP secrets are always protected and can only be accessed upon entry into Secure Mode.
- During Secure Entry Mode all switches are initially set with all controls on (restricted access) Two exceptions are the OTP secrets control (OTPSEN bit) is not accessible so access to the secrets OTP area remains restricted, and the RSTDABL bit remains deactivated (External Reset is allowed).
- During Secure Mode operation all switches are voluntary (initially set) and under the control of authenticated code. Therefore, restricted access controls can be reconfigured by authenticated user code. This includes the activation of Reset Disable (RSTDABL) bit.

The register, shown in Figure 16-6 and Figure 16-7 on page 16-48, is 32-bits wide and requires 32-bit access. Limited write access to a few bits

is allowed in Secure Entry mode, and full write access to all bits is allowed in Secure mode. No write access is allowed in Open Mode.

Secure System Switch Register (SECURE_SYSSWT) Bits 15:0

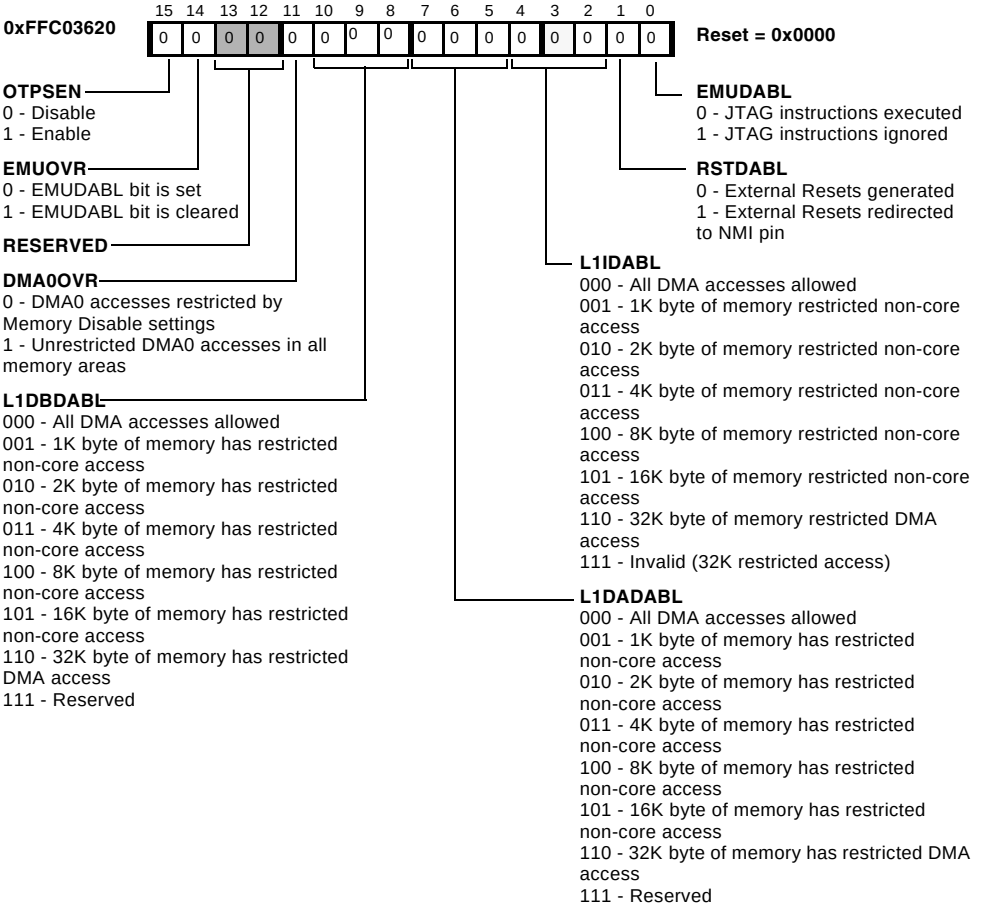

0xFFC03620
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
OTPSEN
0 - Disable
1 - Enable
EMUOVR
0 - EMUDABL bit is set
1 - EMUDABL bit is cleared
RESERVED
DMA0OVR
0 - DMA0 accesses restricted by Memory Disable settings
1 - Unrestricted DMA0 accesses in all memory areas
L1DBDABL
000 - All DMA accesses allowed
001 - 1K byte of memory has restricted non-core access
010 - 2K byte of memory has restricted non-core access
011 - 4K byte of memory has restricted non-core access
100 - 8K byte of memory has restricted non-core access
101 - 16K byte of memory has restricted non-core access
110 - 32K byte of memory has restricted DMA access
111 - Reserved
EMUDABL
0 - JTAG instructions executed
1 - JTAG instructions ignored
RSTDABL
0 - External Resets generated
1 - External Resets redirected to NMI pin
L1IDABL
000 - All DMA accesses allowed
001 - 1K byte of memory restricted non-core access
010 - 2K byte of memory restricted non-core access
011 - 4K byte of memory restricted non-core access
100 - 8K byte of memory restricted non-core access
101 - 16K byte of memory restricted non-core access
110 - 32K byte of memory restricted DMA access
111 - Invalid (32K restricted access)
L1DADABL
000 - All DMA accesses allowed
001 - 1K byte of memory has restricted non-core access
010 - 2K byte of memory has restricted non-core access
011 - 4K byte of memory has restricted non-core access
100 - 8K byte of memory has restricted non-core access
101 - 16K byte of memory has restricted non-core access
110 - 32K byte of memory has restricted DMA access
111 - Reserved

Figure 16-6. Secure System Switch Register, Bits 15:0

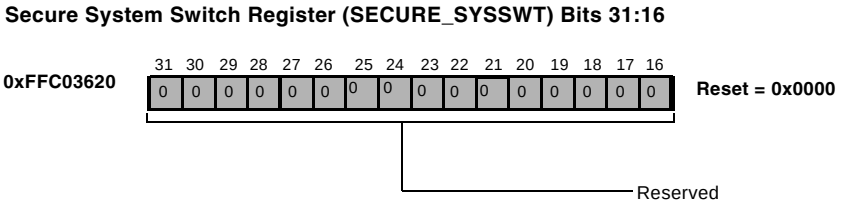


Figure 16-7. Secure System Switch Register, Bits 31:16

Table 16-5. Secure System Switch Register

Bit Position	Bit Name	Bit Description
		Reset = 0x0000 Secured Entry = 0x0007 04D9 Secure Mode = 0x0007 04DB
0	EMUDABL	Emulation Disable. Upon Secured Entry the EMUDABL setting is based on the previous state of EMUOVR. Upon re-entering Open Mode, EMUDABL is cleared. This bit is always read accessible. This bit is write accessible only in Secure Mode. 0 - Analog Devices JTAG emulation instructions are recognized and executed. Once this bit is cleared while in Secure Mode it will not be set upon Secured Entry. This condition will remain until reset, at which time it is cleared. This feature is used in security debug. 1 - Analog Devices JTAG emulation instructions are ignored. Standard emulation commands such as bypass are allowed.
1	RSTDABL	Reset Disable. This bit is not effected upon Secured Entry. This bit is set upon entering Secure Mode. Upon re-entering Open Mode, RSTDABL is cleared. This bit is always read accessible. This bit is write accessible only in Secure Mode. 0 - External Resets are generated and serviced normally. 1 - External Resets are redirected to the NMI pin. This avoids circumventing memory clean operations.

Table 16-5. Secure System Switch Register (Continued)

Bit Position	Bit Name	Bit Description
4:2	L1IDABL	L1 Instruction Memory Disable. Upon Secured Entry L1IDABL is set to 0x6. Upon re-entering Unsecure Mode, L1IDABL is cleared. These bits are always read accessible. These bits are write accessible only in Secure Mode. In the event a DMA access is performed to a restricted memory area a DMA memory access error will occur resulting in a DMA_ERR interrupt and a clearing of DMA_RUN. 000 - All DMA accesses are allowed to L1 Instruction areas. 001 - 1K byte of memory (0xFFA0 0000 - 0xFFA0 03FF) has restricted non core access 010 - 2K byte of memory (0xFFA0 0000 - 0xFFA0 07FF) has restricted non core access 011 - 4K byte of memory (0xFFA0 0000 - 0xFFA0 0FFF) has restricted non core access 100 - 8K byte of memory (0xFFA0 0000 - 0xFFA0 1FFF) has restricted non core access 101 - 16K byte of memory (0xFFA0 0000 - 0xFFA0 3FFF) has restricted non core access 110 - 32K byte of memory (0xFFA0 0000 - 0xFFA0 7FFF) has restricted DMA access. This is the initial setting upon entering Secured Entry. 111 - Reserved

Table 16-5. Secure System Switch Register (Continued)

Bit Position	Bit Name	Bit Description
7:5	L1DADABL	L1 Data Bank A Memory Disable. Upon Secured Entry L1DADABL is set to 0x6. Upon re-entering Open Mode, L1DADABL is cleared. These bits are always read accessible. These bits are write accessible only in Secure Mode. In the event a DMA access is performed to a restricted memory area a DMA memory access error will occur resulting in a DMA_ERR interrupt and a clearing of DMA_RUN. 000 - All DMA accesses are allowed to L1 data bank A areas. 001 - 1K byte of memory (0xFF80 0000 - 0xFF80 03FF) has restricted non core access 010 - 2K byte of memory (0xFF80 0000 - 0xFF80 07FF) has restricted non core access 011 - 4K byte of memory (0xFF80 0000 - 0xFF80 0FFF) has restricted non core access 100 - 8K byte of memory (0xFF80 0000 - 0xFF80 1FFF) has restricted non core access 101 - 16K byte of memory (0xFF80 0000 - 0xFF80 3FFF) has restricted non core access 110 - 32K byte of memory (0xFF80 0000 - 0xFF80 7FFF) has restricted DMA access. This is the initial setting upon entering Secured Entry. 111 - Reserved

Table 16-5. Secure System Switch Register (Continued)

Bit Position	Bit Name	Bit Description
10:8	L1DBDABL	L1 Data Bank B Memory Disable. Upon Secured Entry L1DBDABL is set to 0x4 giving L1 Data Bank B 8 Kbyte of non core restricted access. Upon re-entering Open Mode, L1DBDABL is cleared. These bits are always read accessible. These bits are write accessible only in Secure Mode. In the event a DMA access is performed to a restricted memory area a DMA memory access error will occur resulting in a DMA_ERR interrupt and a clearing of DMA_RUN. 000 - All DMA accesses are allowed to L1 data bank B areas. This is the initial setting upon entering Secured Entry. 001 - 1K byte of memory (0xFF90 0000 - 0xFF90 03FF) has restricted non core access 010 - 2K byte of memory (0xFF90 0000 - 0xFF90 07FF) has restricted non core access 011 - 4K byte of memory (0xFF90 0000 - 0xFF90 0FFF) has restricted non core access 100 - 8K byte of memory (0xFF90 0000 - 0xFF90 1FFF) has restricted non core access. This is the initial setting upon entering Secured Entry. 101 - 16K byte of memory (0xFF90 0000 - 0xFF90 3FFF) has restricted non core access 110 - 32K byte of memory (0xFF90 0000 - 0xFF90 7FFF) has restricted DMA access. 111 - Reserved
11	DMA0OVR	DMA0 Memory Access Override Entering Secured Entry or Secure Mode does not effect this bit. Upon re-entering Open Mode, DMA0OVR is cleared. This bit is always read accessible. This bit is write accessible in both Secured Entry and Secure Mode. Controls DMA0 access to L1 Instruction, L1 Data and memory other than L1 regions. When clear access restrictions are based on Memory Disable settings within this register. 0 - DMA0 accesses are restricted based on Memory Disable settings. 1 - Unrestricted DMA0 accesses are allowed to all memory areas.
12	Reserved	Reserved bit. This reserved bit always returns a “0” value on a read access. Writing this bit with any value has no effect.
13	Reserved	Reserved bit. This reserved bit always returns a “0” value on a read access. Writing this bit with any value has no effect.

Table 16-5. Secure System Switch Register (Continued)

Bit Position	Bit Name	Bit Description
14	EMUOVR	Emulation Override This bit is always read accessible. This bit may be written with a 1 in Secure Mode only. This bit can be cleared in any mode (Open Mode, Secured Entry and Secure mode). Controls the value of EMUDABL upon Secured Entry. 0 - Upon Secured Entry the EMUDABL bit is set. 1 - Upon Secured Entry the EMUDABL bit is cleared. This bit can only be set when EMUDABL (bit-0) is written with a "0" while this bit (bit-14) is simultaneously written with a 1.
15	OTPSEN	OTP Secrets Enable. This bit can be read in all modes but is write accessible in Secure Mode only. 0 - Read and Programming access of the "secured" OTP Fuse area is restricted. Accesses will result in an access error (FERROR) 1 - Read and Programming access of the "secured" OTP Fuse area is allowed. If the corresponding program protection bit for an access is set, a program access is protected regardless of this bit's setting.
31:16	Reserved	Reserved. To ensure upward compatibility with future implementations, write back the value that is read from these bits.

Secure Control (SECURE_CONTROL) Register

The SECURE_CONTROL register is used during Secure Entry Mode authentication. This register is used to establish Secure Mode transition and can be

used at any time to exit from Secure Mode. The register, shown in Figure 16-8, is 16-bits wide and requires 16-bit access.

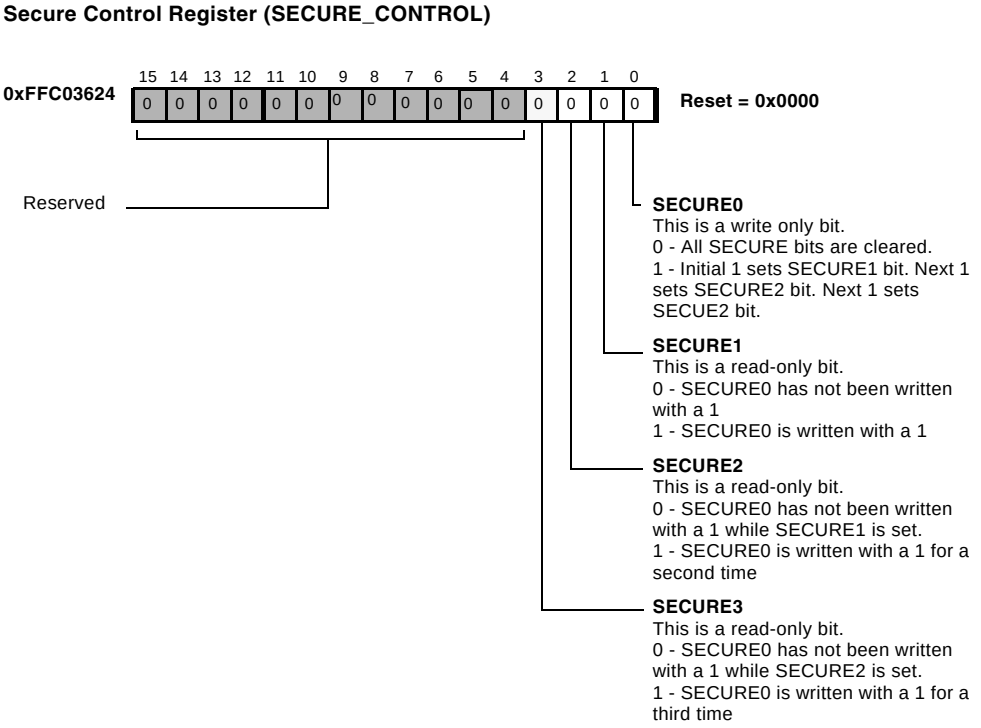

Secure Control Register (SECURE_CONTROL)
0xFFC03624
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Reserved
SECURE0
This is a write only bit.
0 - All SECURE bits are cleared.
1 - Initial 1 sets SECURE1 bit. Next 1 sets SECURE2 bit. Next 1 sets SECUE2 bit.
SECURE1
This is a read-only bit.
0 - SECURE0 has not been written with a 1
1 - SECURE0 is written with a 1
SECURE2
This is a read-only bit.
0 - SECURE0 has not been written with a 1 while SECURE1 is set.
1 - SECURE0 is written with a 1 for a second time
SECURE3
This is a read-only bit.
0 - SECURE0 has not been written with a 1 while SECURE2 is set.
1 - SECURE0 is written with a 1 for a third time

Figure 16-8. Secure Control Register

Table 16-6. Secure Control Register

Bit Position	Bit Name	Bit Description
		Reset = 0x0000
0	SECURE0	SECURE 0 This is a write only bit. A read always returns “0”. A 1 value can only be written to SECURE0 when in Secured Entry. The purpose of this control bit is to require 3 successive writes with a value of 1 to SECURE0 in order to enter Secure Mode. 0 - When written with a “0” value, all SECURE bits within this register are cleared and Open Mode is entered. All SYSSWT bits are cleared with the exception of EMUOVR. If EMUOVR had been set by the user, it will remain set (until RESET is asserted or until it is written with a “0”). 1 - Initially when written with a 1 value SECURE1 is set. With a subsequent 1 written SECURE2 is set. A subsequent 1 written will set SECURE3. Upon a set of SECURE3 Secure Mode is entered.
1	SECURE1	SECURE 1 This is a read-only bit and indicates a successful write of SECURE0 with a data value of 1 0 - SECURE0 has not been written with a 1 value 1 - SECURE0 is written with a 1 value
2	SECURE2	SECURE 2 This is a read-only bit and indicates two successful writes of SECURE0 with a data value of 1 has occurred 0 - SECURE0 has not been written with a 1 value while SECURE1 was set. 1 - SECURE0 is written with a 1 value for a second time.
3	SECURE3	SECURE 3 This is a read-only bit and indicates three successful writes of SECURE0 with a data value of 1 has occurred. 0 - SECURE0 has not been written with a 1 value while SECURE2 was set 1 - SECURE0 is written with a 1 value for a third time. The part is currently in Secure Mode and the SYSSWT register is writable by Authenticated code.

SECURE0 bit is user accessible and is used to exit from Secure Mode. Bits SECURE1, SECURE2, and SECURE3 are not user accessible and are accessed only by the firmware during the digital signature validation process.

Secure Status (SECURE_STATUS) Register

The SECURE_STATUS register provides information about the current secure state. This information can be used during security mode control as well

as understanding why an authentication attempt has failed. The register, shown in Figure 16-9, is 16-bits wide and requires 16-bit access.

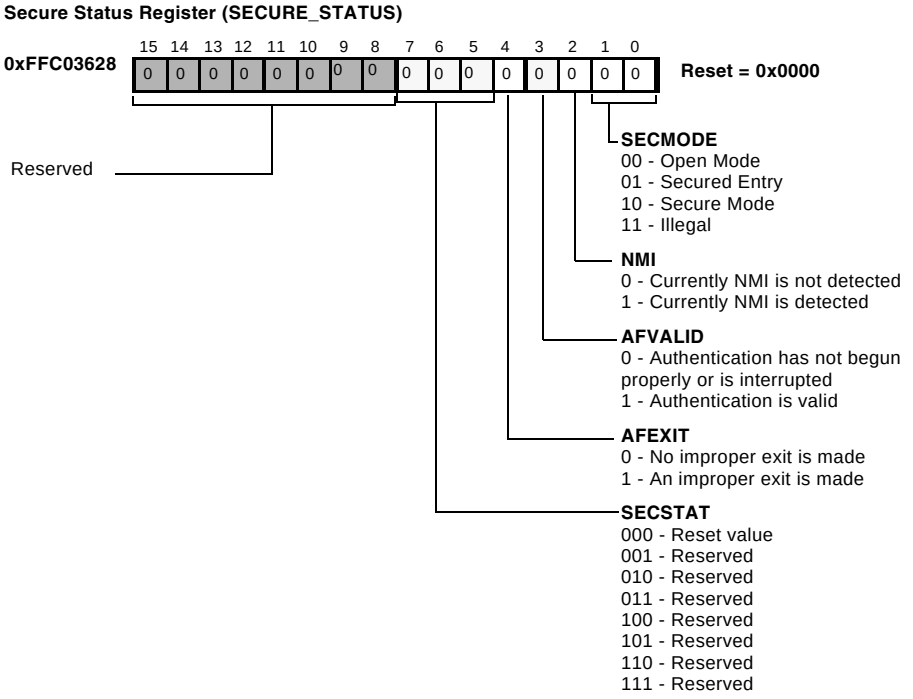

Secure Status Register (SECURE_STATUS)
0xFFC03628
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Reserved
SECMODE
00 - Open Mode
01 - Secured Entry
10 - Secure Mode
11 - Illegal
NMI
0 - Currently NMI is not detected
1 - Currently NMI is detected
AFVALID
0 - Authentication has not begun properly or is interrupted
1 - Authentication is valid
AFEXIT
0 - No improper exit is made
1 - An improper exit is made
SECSTAT
000 - Reset value
001 - Reserved
010 - Reserved
011 - Reserved
100 - Reserved
101 - Reserved
110 - Reserved
111 - Reserved

Figure 16-9. Secure Status Register

Table 16-7. Secure Status Register

Bit Position	Bit Name	Bit Description
		Reset = 0x0000
1:0	SECMODE	Secure Mode Control State These are read-only bits that reflects the current Secure Mode Control's state. 00 - Open Mode 01 - Secured Entry 10 - Secure Mode 11 - Illegal
2	NMI	This is a read-only bit that reflects the detection of NMI. 0 - Currently NMI is not detected. 1 - Currently NMI is detected.
3	AFVALID	Authentication Firmware Valid This is a read-only bit that reflects the state of the hardware monitor logic. If execution of authentication has begun properly and has had uninterrupted operation the authentication is considered valid. A valid authentication is required for Secured Entry and Secure Mode operation. 0 - Authentication has not begun properly or is interrupted. 1 - Authentication is valid and is progressing properly and uninterrupted.
4	AFEXIT	Authentication Firmware Exit This is a write one to clear status bit. In the event authentication has begun properly but has had an improper exit before completion, this bit is set. This can only occur on an exit from Secured Entry back to Open Mode. 0 - No improper exit is made while executing authentication firmware. 1 - An improper exit from authentication firmware is made.
7:5	SECSTAT	Secure Status These are some read write bits which is defined later. These are intended to pass a status back to the handler in the event an authentication has failed. 000 - Reset value 001 - Reserved 010 - Reserved 011 - Reserved 100 - Reserved 101 - Reserved 110 - Reserved 111 - Reserved

Authentication Firmware Valid (AFVALID) is an input to the Secure State Machine and not an output control/status. AFVALID goes active based on reaching the correct program counter address.

17 SYSTEM RESET AND BOOTING

This document contains material that is subject to change without notice. The content of the boot ROM as well as hardware behavior may change across silicon revisions. See the anomaly list for differences between silicon revisions.

Overview

When the $\overline{\text{RESET}}$ input signal releases, the processor starts fetching and executing instructions from the on-chip boot ROM at address 0xEF00 0000.

The internal boot ROM includes a small boot kernel that loads application data from an external memory or host device. The application data is expected to be available in a well-defined format called the boot stream. A boot stream consists of multiple blocks of data and special commands that instruct the boot kernel how to initialize on-chip L1 memories as well as off-chip volatile memories.

The boot kernel processes the boot stream block-by-block until it is instructed by a special command to terminate the procedure and jump to the application's programmable start address, which traditionally is at 0xFFA0 0000 in on-chip L1 memory. This process is called "booting."

The processor features four dedicated input pins BMODE[3:0] that select the booting mode. The boot kernel evaluates the BMODE pins and performs

booting from respective sources. Table 17-1 describes the modes of the BMODE pins.

Table 17-1. Booting Modes

BMODE[3:0]	Boot Source	Description
0000	No boot – idle	The processor does not boot. Rather, the boot kernel executes an IDLE instruction.
0001	Boot from 8-bit or 16-bit external flash memory	The kernel boots from address 0x2000 0000 in asynchronous memory bank 0. The first byte of the boot stream contains further instructions whether the memory is eight or 16 bits wide.
0010	Boot from 16-bit asynchronous FIFO	By using the handshaked memory DMA (HMDMA1) feature through the $\overline{\text{DMAR1}}$ input, the kernel boots from address 0x2030 0000 in asynchronous memory bank 3.
0011	Boot from serial SPI memory	After an initial device detection routine, the kernel boots from either 8-bit, 16-bit, 24-bit or 32-bit addressable SPI flash or EEPROM memory that connects to $\overline{\text{SPI\_SEL1}}$.
0100	Boot from SPI host	In this slave mode, the kernel expects the boot stream to be applied to SPI by an external host device.
0101	Boot from serial TWI memory	The kernel boots from TWI memory connected to TWI. Memory is expected to respond to the unique slave identifier of 0xA0.
0110	Boot from TWI host	In this slave mode, the kernel expects the boot stream to be applied to TWI by an external host device. The Blackfin processor uses the slave identifier 0x5F.

Table 17-1. Booting Modes (Continued)

BMODE[3:0]	Boot Source	Description
0111	Boot from UART0 host	In this slave mode, the kernel expects the boot stream to be applied to UART0 by an external host device. Prior to providing the boot stream, the host device is expected to send a 0x40 (ASCII '@') character that is examined by the kernel to adjust the bit rate.
1000	Boot from UART1 host.	Same as BMODE = 0111 but UART1 interface is used.
1001	Reserved	
1010	Boot from SDRAM memory[1]	This mode provides a quick warm boot option. It requires the SDRAM controller to be programmed by the preboot routine based on OTP settings. The kernel starts booting from address 0x0000 0010.
1011	Boot from on-chip OTP memory	This is the only stand-alone booting mode. It boots from the on-chip serial OTP memory. By default, the boot stream is expected to reside from OTP page 0x40 on. The start page can be altered by programming the OTP_START_PAGE field in OTP page PBS01H.
1100	Boot from 8-bit NAND flash using port F data interface.	The boot kernel automatically detects whether an 8-bit small-page device or an 8-bit large-page device is connected to the NFC. The NAND flash may optionally contain further initialization code that enables some more advanced boot options.
1101	Boot from 8-bit NAND flash using the port H data interface.	The boot kernel automatically detects whether an 8-bit small-page device or an 8-bit large-page device is connected to the NFC. The NAND flash may optionally contain further initialization code that enables some more advanced boot options.

Table 17-1. Booting Modes (Continued)

BMODE[3:0]	Boot Source	Description
1110	Boot from 16-bit Host DMA	The kernel initializes the Host DMA unit to 16-bit ACK mode. Boot stream parsing is up to the host device. An HIRQ command causes the kernel to issue a CALL to the address 0xFFA0 0000.
1111	Boot from 8-bit Host DMA	The kernel initializes the Host DMA unit 8-bit INT mode. Boot stream parsing is up to the host device. An HIRQ command causes the kernel to issue a CALL to the address 0xFFA0 0000.

1 This chapter uses the term SDRAM as a synonym for off-chip synchronous dynamic memory. For the ADSP-BF522/523/524/525/526/527 products, this includes the Mobile SDRAM standard.

Reset and Power-up

There is a subroutine in the boot kernel known as "preboot", which is executed prior to the boot mode being processed. This preboot routine can customize default values of MMR registers, such as the PLL and SDRAM controller registers. Furthermore, the SPI and TWI master modes can be customized. The preboot behavior is controlled through OTP programming.

To enable booting from volatile memories such as SDRAM, the SDRAM controller must be programmed *before* data can be loaded into the memory. Either the preboot or the initialization code mechanism can be used for this purpose.

Table 17-2 describes the six types of resets.

Each type resets the core except for the System Software reset.

Table 17-2. Resets

Reset	Source	Result
Hardware reset	The RESET pin causes a hardware reset.	Resets both the core and the peripherals, including the dynamic power management controller (DPMC). Resets bits [15:4] of the SYSCR register. For more information, see "System Reset Configuration (SYSCR) Register" on page 17-100.
Wake up from hibernate state	Wake-up event as enabled in the VR_CTL register and reported by the PLL_STAT register.	Behaves as hardware reset except the WURESET bit in the SYSCR register is set. Booting can be performed conditionally on this event.
System software reset	Calling the bfrom_SysControl() routine with the SYSCTRL_SYSRESET option triggers a system reset.	Resets only the peripherals, excluding the RTC (real time clock) block and most of the DPMC. The system software reset clears bits [15:13] and bits [11:4] of the SYSCR register, but not the WURESET bit. The core is not reset and a boot sequence is not triggered. Sequencing continues at the instruction after bfrom_SysControl() returns.
Watchdog timer reset	Programming the watchdog timer causes a watchdog timer reset.	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. (Because of the partial reset to the DPMC, the watchdog timer reset is not functional when the processor is in Sleep or Deep Sleep modes.) The SWRST or the SYSCR register can be read to determine whether the reset source was the watchdog timer.

Table 17-2. Resets (Continued)

Reset	Source	Result
Core double-fault reset	A core double fault occurs when an exception happens while the exception handler is executing. If the core enters a double-fault state, a reset can be caused by unmasking the DOUBLE_FAULT bit in the SWRST register.	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The SWRST or SYSCR registers can be read to determine whether the reset source was a core double-fault.
Software reset	This reset is caused by executing a RAISE 1 instruction or by setting the software reset (SYSRST) bit in the core debug control register (DBGCTL) through emulation software through the JTAG port. The DBGCTL register is not visible to the memory map.	Program executions vector to the 0xEF00 0000 address. The boot code executes an immediate system reset to ensure system consistency.

Hardware Reset

The processor chip reset is an asynchronous reset event. The $\overline{\text{RESET}}$ input pin must be deasserted after a specified asserted hold time to perform a hardware reset. For more information, see the product data sheet.

A hardware-initiated reset results in a system-wide reset that includes both core and peripherals. After the RESET pin is deasserted, the processor ensures that all asynchronous peripherals have recognized and completed a reset. After the reset, the processor transitions into the boot mode sequence configured by the state of the BMODE pins.

The BMODE pins are dedicated mode control pins. No other functions are shared with these pins, and they may be permanently strapped by tying them directly to either V_{DDEXT} or GND. The pins and the corresponding bits in the SYSCR register configure the boot mode that is employed after hardware reset or system software reset. See the Blackfin Processor Programming Reference for further information.

Software Resets

A software reset may be initiated in three ways.

- By the watchdog timer, if appropriately configured
- Calling the `bfrom_SysControl()` API function residing in the on-chip ROM. For further information, see Chapter 18, "Dynamic Power Management".
- By the `RAISE 1` instruction

The watchdog timer resets both the core and the peripherals, as long as the processor is in Active or Full-On mode. A system software reset results in a reset of the peripherals without resetting the core and without initiating a booting sequence.

In order to perform a system reset, the `bfrom_SysControl()` routine must be called while executing from L1 memory (either as cache or as SRAM). When L1 instruction memory is configured as cache, make sure the system reset sequence is read into the cache.

After either the watchdog or system software reset is initiated, the processor ensures that all asynchronous peripherals have recognized and completed a reset.

For a reset generated by formatting the watchdog timer, the processor transitions into the boot mode sequence. The boot mode is configured by the state of the `BMODE` bit field in the `SYSCR` register.

A software reset is initiated by executing the `RAISE 1` instruction or setting the software reset (`SYSRST`) bit in the core debug control register (`DBGCTL`) (`DBGCTL` is not visible to the memory map) through emulation software through the JTAG port.

A software reset only affects the state of the core. The boot kernel immediately issues a system reset to keep consistency with the system domain.

Reset Vector

When reset releases, the processor starts fetching and executing instructions from address 0xEF00 0000. This is the address where the on-chip boot ROM resides.

On a hardware reset, the boot kernel initializes the EVT1 register to 0xFFA0 0000. When the booting process completes, the boot kernel jumps to the location provided by the EVT1 vector register. With the exception of the HOSTDP boot modes, the content of the EVT1 register is overwritten by the TARGET ADDRESS field of the first block of the applied boot stream. If the BCODE field of the SYSCR register is set to 1 (no boot option), the EVT1 register is not modified by the boot kernel on software resets. Therefore, programs can control the reset vector for software resets through the EVT1 register. This process is illustrated by the flow chart in Figure 17-1.

The content of the EVT1 register may be undefined in emulator sessions.

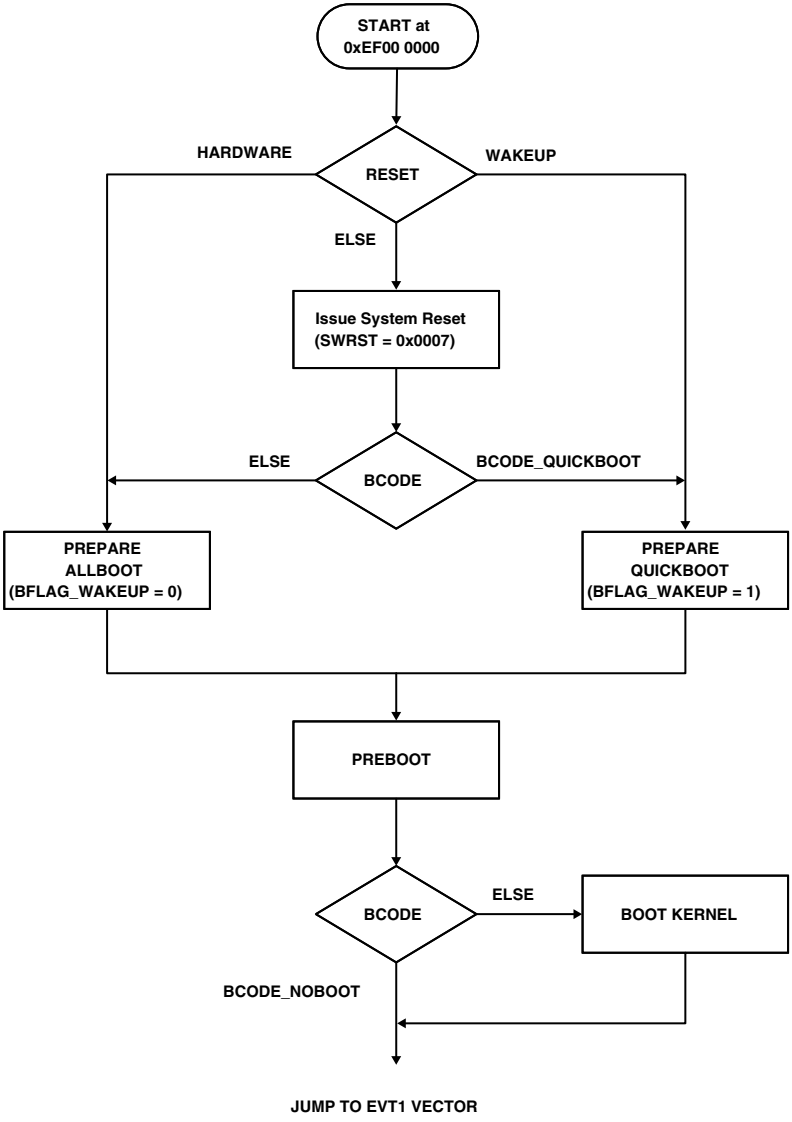


Figure 17-1. Global Boot Flow

Servicing Reset Interrupts

The processor services a reset event like other interrupts. The reset interrupt has top priority. Only emulation events have higher priority. When coming out of reset, the processor is in supervisor mode and has full access to all system resources. The boot kernel can be seen as part of the reset service routine. It runs at the top interrupt priority level.

Even when the boot process has finished and the boot kernel passes control to the user application, the processor is still in the reset interrupt. To enter user mode, the reset service routine must initialize the `RETI` register and terminate with an `RTI` instruction.

For a programming example, see "System Reset" on page 17-138.

Listing 17-1 and Listing 17-2 on page 17-138 show code examples that handle the reset event. See the Blackfin Processor Programming Reference for details on user and supervisor modes.

Systems that do not work in an operating system environment may not enter user mode. Typically, the interrupt level needs to be degraded down to IVG15. Listing 17-3 and Listing 17-4 on page 17-139 show how this is accomplished.

Since the boot kernel is running at reset interrupt priority, NMI events, hardware errors and exceptions are not serviced at boot time. As soon as the reset service routine returns, the processor can service the events that occurred during the boot sequence. It is recommended that programs install NMI, hardware error, and exception handlers before leaving the reset service routine. This includes proper initialization of the respective event vector registers `EVTx`.

Preboot

After reset, the boot kernel residing in the on-chip boot ROM does not immediately start processing the boot stream. First it calls a subroutine called preboot, as shown in Figure 17-2 on page 17-12 and Figure 17-3 on page 17-13. The preboot routine customizes the default values of several system MMR registers based on user-configurable OTP (one-time programmable) memory. The following modules can be customized in this way.

- PLL and voltage regulator settings
- SDRAM controller settings
- Asynchronous EBIU settings

Some OTP bits customize the boot process:

- Bit rate of SPI and TWI master boot modes
- TWI master boot addressing scheme
- Activation of SPI fast read mode
- Boot host wait (`HWAIT`) signal

Further OTP bits let the user disable certain features of the processor:

- Individual boot modes (for security reasons)

Finally, certain bits are already preset in the factory:

- USB voltage trim
- Individual boot modes

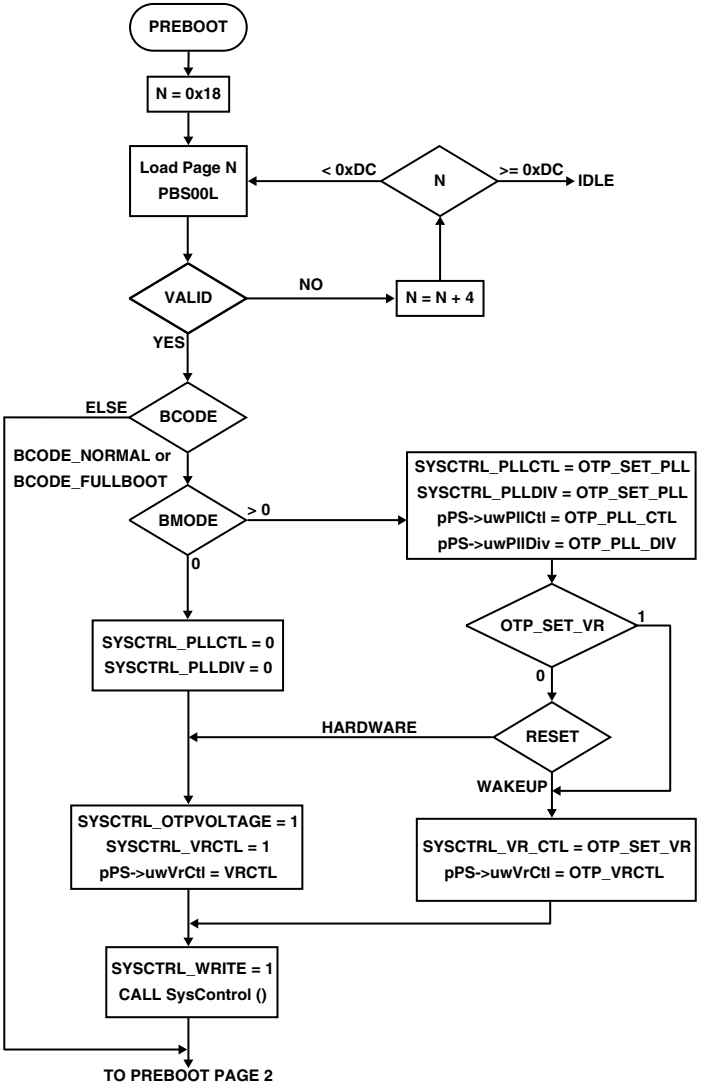


Figure 17-2. Preboot Flow 1 of 2

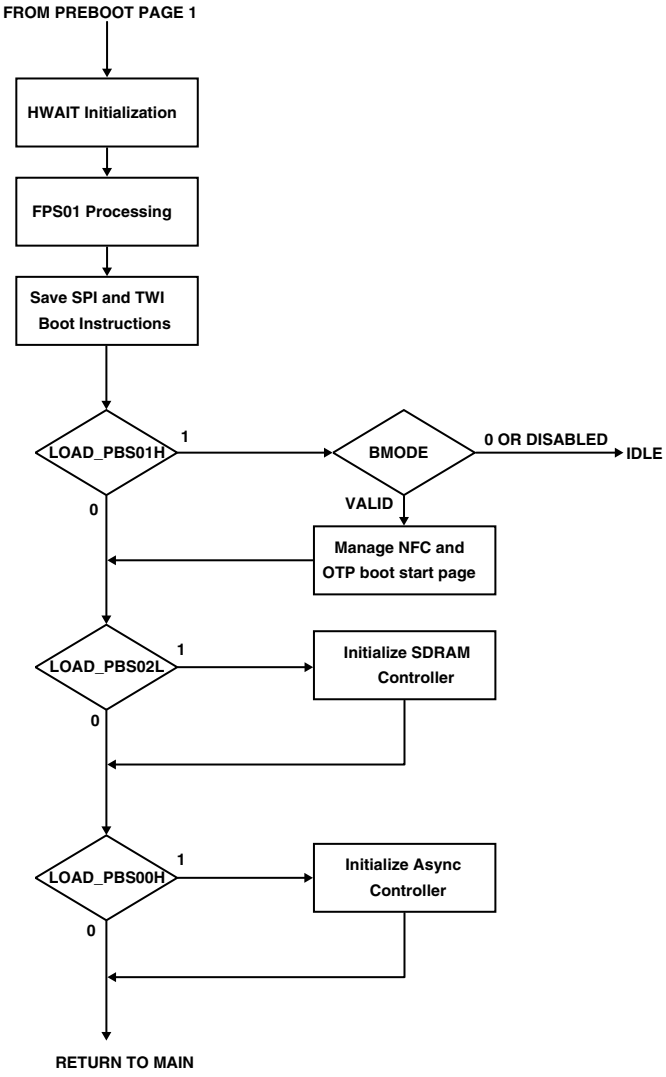


Figure 17-3. Preboot Flow 2 of 2

Factory Page Settings (FPS)

The content of the boot ROM is identical across all ADSP-BF52x Blackfin processors. The factory settings prevent the boot ROM from accidentally accessing resources that are not present on a given processor, which would result in unpredictable behavior and/or hardware errors. The boot kernel goes to a safe idle state when the user configures the `BMODE` pins to a boot mode that is not available on a specific part.

For this purpose, the preboot routine always reads the `FPS01L` and `FPS01H` half pages from OTP memory. These half pages contain factory trim values for the USB PHY controller that are managed at preboot time, as required. In addition, the `bfrom_SysControl()` routine reads the half page `FPS04H` to apply factory trim values to the voltage regulator.

Preboot Page Settings (PBS)

Four OTP pages optionally enable the user to customize the behavior of the processor immediately after reset. These four pages (eight half pages) can be seen as one contiguous pre-boot settings (PBS) block. By default, the block spans OTP pages 0x18 to 0x1B. The OTP pages serve the following purposes:

- PBS00L (by default, on half page 0x18L, see "Lower PBS00 Half Page" on page 17-106 for details)

  PLL and voltage regulator settings

  Boot customization

  Instruction whether to load further half pages

- PBS00H (by default, on half page 0x18H, see "Upper PBS00 Half Page" on page 17-110 for details)

Asynchronous EBIU register settings

- PBS01L (by default on half page 0x19L)

  Reserved

- PBS01H (by default, on half page 0x19H, see "Upper PBS01 Half Page" on page 17-111 for details)

  Disabling of boot modes

  NFC controller register settings

  OTP boot start page

- PBS02L (by default, on half page 0x1AL, see "Lower PBS02 Half Page" on page 17-114 for details)

  Synchronous EBIU register settings

- PBS02H (by default, on half page 0x1AH, see "Upper PBS02 Half Page" on page 17-115 for details)

  Reserved in current silicon revision. Do not use.

- PBS03L (by default, on half page 0x1BL, see "Reserved Half Pages" on page 17-115 for details)

  Reserved in current silicon revision. Do not use.

- PBS03H (by default, on half page 0x1BH, see "Reserved Half Pages" on page 17-115 for details)

  Reserved in current silicon revision. Do not use.

The preboot routine reads the main page `PBS00L` first. Since this page may instruct the preboot routine to alter the PLL settings, further pages may read more quickly. This page also instructs the preboot whether further OTP half pages have to be loaded and processed. By default, the `PBS00L` page reads all zeroes, and the preboot does not load further PBS pages.

Alternative PBS Pages

Especially during the development cycle, the user may fail to write the proper value to OTP memory and may make multiple attempts to get things right. Therefore, the PBS00L page provides a mechanism to invalidate the entire PBS block (consisting of pages (0x18, 0x19, 0x1A and 0x1B) and to use pages 0x1C to 0x1F instead. To do so, set the two OTP_INVALID bits (bits 62 and 63 on the PBS00L page). If both bits are set, the preboot routine disregards potential error codes returned by the bfrom_OtpRead() routine and continues processing from page 0x1C on. The active PBS block now spans the pages 0x1C to 0x1F. If the user wants to invalidate the second set of OTP pages as well, setting bits 62 and 63 on page 0x1C (which is the new PBS00L half page) instructs the preboot routine to continue at page 0x20, and so on.

Theoretically, this can be repeated up to page 0xD8L, if the pages are not required for other purposes. There are 49 chances to get things right, before a device may become useless. Note that every page that needs to be read by the preboot routine causes additional delay to the boot process.

Programming PBS Pages

Due to the need for error checking and correction (ECC), a 64-bit OTP half page must be written all at once. It is recommended that PBS pages be programmed only through the API function bfrom_OtpWrite().

If it is anticipated that the user is customizing the boot-related OTP pages for safety or security reasons, it is recommended that all PBS blocks be locked at production time to protect these pages from being tampered with in the field.

Reading OTP memory is subject to a potential failure rate. Since the preboot only accesses OTP memory through the bfrom_OtpRead() function, the ECC error correction is applied and the statistical failure rate is very low. However, the way the PBS00L page is tested for being invalid may at some point reduce the ECC

reliability. To keep failure rates at a minimum, it is a good idea to duplicate the content of pages 0x18–0x1B on pages 0x1C–0x1F. For production parts, the final block should be followed by its exact copy to maintain the lowest failure rates. Then, even the unlikely case where one of the OTP_INVALID bits is read incorrectly would not cause the boot to fail.

Recovering From Misprogrammed PBS Pages

The preboot mechanism provides a powerful method to customize the chip to the needs of the user. However, as a downside, there are chances that invalid values programmed to the PBS pages prevent the processor from operating within required operating conditions. There is specific risk when the PLL and the voltage regulator are programmed with meaningless values during the development cycle.

In such cases, the boot mode BMODE = b#0000 helps. In this mode, the preboot routine does not attempt to read any of the user-programmable PBS pages, and the boot kernel does not try to boot any data. Rather, the processor is idled immediately after the FPS pages have been processed. Using the in-circuit emulator, the user then has the option to invalidate the actual PBS settings by overwriting both OTP_INVALID bits in the actual PBS00L with 1s.

For safety reasons, none of the boot modes, except the emulator, can get control over the processor when in this state.

Customizing Power Management

When the processor awakes with default PLL and voltage regulator settings, the preboot mechanism can be used to alter these settings to custom values before the boot process takes place. This is done by programming the OTP half page PBS00L.

If the OTP_SET_PLL bit is programmed to a 1, the value in the OTP_PLL_DIV bit field is copied into the PLL_DIV register, and the OTP_PLL_CTL bit field

is copied into the PLL_CTL register, followed by the required IDLE instruction (if the contents of PLL_CTL are being altered).

If the OTP_SET_VR bit is programmed to a 1, the value in the OTP_VR_CTL bit field is copied into the VR_CTL register, followed by the required IDLE instruction (if the contents of VR_CTL are being altered).

The preboot mechanism invokes the bfrom_SysControl() routine to alter the PLL and the voltage regulator. The bfrom_SysControl() routine not only performs custom instructions, it also applies correction values from factory OTP pages FPS01 and FPS04. See Chapter 18, "Dynamic Power Management" for details on the bfrom_SysControl() routine.

Customizing Booting Options

The OTP pages accessible by the preboot mechanism can also be used to customize some of the booting options. For example:

- TWI master boot mode operating frequency
- SPI master boot mode operating frequency
- SPI master boot mode read operation mode
- Start page for OTP boot mode
- HWAIT signal behavior
- Disabling of unwanted boot modes

In TWI master boot mode, the OTP_TWI_PRESCALE and OTP_TWI_CLKDIV values in the preboot half page PBS00L control the respective prescale and clock divider values written to the TWI_CONTROL and TWI_CLKDIV registers. The table of values can be found in "TWI Master Boot Mode" on page 17-72. The bit field OTP_TWI_TYPE controls whether one, two, three or four address bytes are used to address the I^2C memory device. By

default, two address bytes are used. The address bits embedded in the read command are not counted.

In SPI master boot mode, the OTP_SPI_BAUD register in the preboot half page PBS00L controls the value written to the SPI_BAUD registers. By default, the clock divider value of 133 can be reduced in power-of-two steps. The table of values can be found in "SPI Master Boot Modes" on page 17-65. The OTP_SPI_BAUD bit instructs the boot kernel to use the 0x0B SPI read command instead of the normal 0x03 read command when accessing the SPI memory device.

In OTP boot mode, the boot kernel normally assumes that the boot stream starts at OTP page 0x40L. The user can change this start page by programming the OTP_START_PAGE bit field in the preboot half page PBS01H.

The boot host wait (HWAIT) signal is available in all boot modes. If the OTP_RESETOUT_HWAIT bit in the preboot half page PBS00L is set, the boot kernel does not toggle HWAIT. Rather, it simply drives it to simulate a reset output signal.

If safety or security of an application is impacted by the existence of certain boot modes, the boot mode disable bits in preboot half page PBS01H can be used to disable unwanted boot modes. If a disabled boot mode is chosen by the BMODE pins, the boot kernel goes into a safe idle state and stops processing. The half page PBS01H is only loaded when the OTP_LOAD_PBS01H bit in the PBS00L page is set.

Customizing the Asynchronous Port

The preboot half page PBS00H contains instructions to customize the asynchronous portion of the EBIU controller. This half page is only loaded and processed when the OTP_LOAD_PBS00H bit in the PBS00L half page is programmed to a 1.

The OTP_EBIU_AMG field is copied into the EBIU_AMGCTL register. While the lower bit controls the CLKOUT signal, the upper three AMBEN bits control which of the four asynchronous banks are enabled. For the FIFO boot mode, the three AMBEN bits are overruled and are all always set.

The preboot routine analyzes the three AMBEN bits and initializes the 16-bit portions (this routine is similar to the enabled banks in the EBIU_AMBCTL0 and EBIU_AMBCTL1 registers) with the value provided in the 16-bit OTP_EBIU_AMBCTL field. In this way, the bus timing of the asynchronous port can be customized prior to the boot process.

Customizing the Synchronous Port

Since many Blackfin applications require data and/or instruction code to be loaded into the SDRAM memory at boot time, the SDRAM controller must be initialized beforehand. This can be done by using either the "Initialization Code" on page 17-37 or the preboot mechanism described here. For the SDRAM boot mode, only the preboot mechanism is valid.

If the OTP_LOAD_PBS02L bit in the PBS00L half page is programmed to a "1", the preboot half page PBS02L is also loaded and processed to initialize the SDRAM controller.

First, the preboot routine tests the SDRS bit in the EBIU_SDSTAT status register. To avoid reconfiguring an already enabled SDRAM controller, processing is bypassed if this bit is already set.

The lower twelve bits of the OTP_EBIU_SDRCC refresh rate value are written to the EBIU_SDRCC register. Similarly, the lower six bits of the OTP_EBIU_SDBCTL value are written to the EBIU_SDBCTL bank control register. The entire 32-bit OTP_EBIU_SDGCTL word is copied to the EBIU_SDGCTL global SDRAM control register.

To minimize access latencies during the boot process, an initial dummy access to the SDRAM is performed immediately after the SDRAM controller is set up. By default, a 32-bit dummy is read from address 0x0000

0000. If the OTP_EBIU_POWERON_DUMMY_WRITE bit is programmed to a “1”, a 32-bit zero value is written to that address instead. This option takes less time but destroys the previous content of that memory location. Address 0x0000 0000 is rarely used for regular processing since it represents a target of NULL pointers.

Basic Booting Process

Once the preboot routine returns, the boot kernel residing in the on-chip boot ROM starts processing the boot stream. The boot stream is either read from memory or received from a host processor. A boot stream represents the application data and is formatted in a special manner. The application data is segmented into multiple blocks of data. Each block begins with a block header. The header contains control words such as the destination address and data length information.

As Figure 17-4 on page 17-22 illustrates, the VisualDSP++ tools suite features a loader utility (`elfloader.exe`). The loader utility parses the input executable file (`.DXE`), segments the application data into multiple blocks, and creates the header information for each block. The output is stored in a loader file (`.LDR`). The loader file contains the boot stream and is made available to hardware by programming or burning it into non-volatile

external memory. Refer to the *VisualDSP++ Loader Manual* for information on switches for loader files.

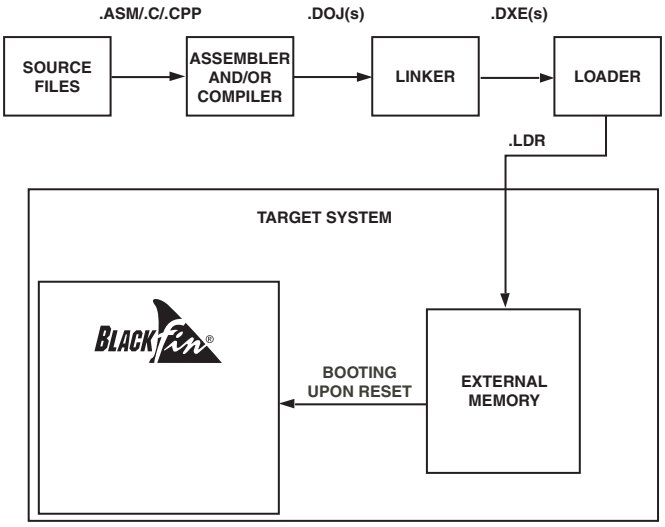


Figure 17-4. Project Flow for a Standalone System

Figure 17-5 on page 17-23 shows the parallel or serial boot stream contained in a flash memory device. In host boot scenarios, the non-volatile memory more likely connects to the host processor rather than directly to the Blackfin processor. After reset, the headers are read and parsed by the on-chip boot ROM, and processed block-by-block. Payload data is copied to destination addresses, either in on-chip L1 memory or off-chip SRAM/SDRAM.

Booting into scratchpad memory (0xFFB0 0000–0xFFB0 0FFF) is not supported. If booting to scratchpad memory is attempted, the processor hangs within the on-chip boot ROM. Similarly, booting into the upper 16 bytes of L1 data bank A (0xFF80 7FF0–0xFF80 7FFF by default) is not supported. These memory locations are used by the boot kernel for intermediate storage of block

header information. These memory regions cannot be initialized at boot time. After booting, they can be used by the application during runtime.

When the `BFLAG_INDIRECT` flag for any block is set, as in TWI boot modes, the boot kernel uses another memory block in L1 data bank B (by default, 0xFF90 7E00–0xFF90 7FFF) for intermediate data storage. To avoid conflicts, the VisualDSP++ `elfloader` utility ensures this region is booted last.

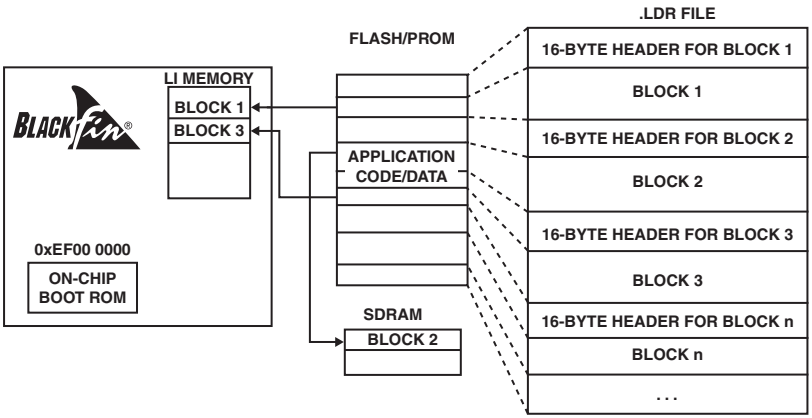


Figure 17-5. Booting Process

The entire source code of the boot ROM is shipped with the VisualDSP++ tools installation. Refer to the source code for any additional questions not covered in this manual. Note that minor maintenance work may be done to the content of the boot ROM when silicon is updated.

Block Headers

A boot stream consists of multiple boot blocks, as shown in Figure 17-5 on page 17-23. Every block is headed by a 16-byte block header. How-

ever, every block does not necessarily have a payload, as shown in Figure 17-6 on page 17-24.

The 16 bytes of the block header are functionally grouped into four 32-bit words, the BLOCK CODE, the TARGET ADDRESS, the BYTE COUNT, and the ARGUMENT fields.

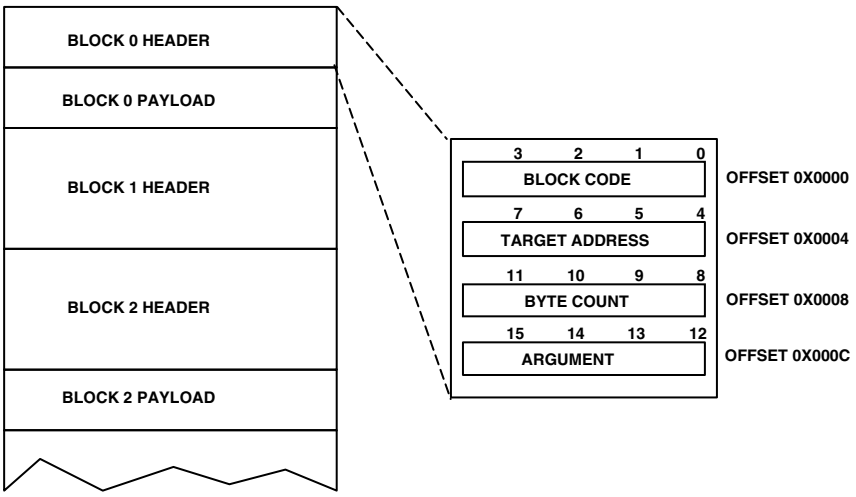


Figure 17-6. Boot Stream Headers

Block Code

The first 32-bit word is the BLOCK CODE. See Figure 17-7.

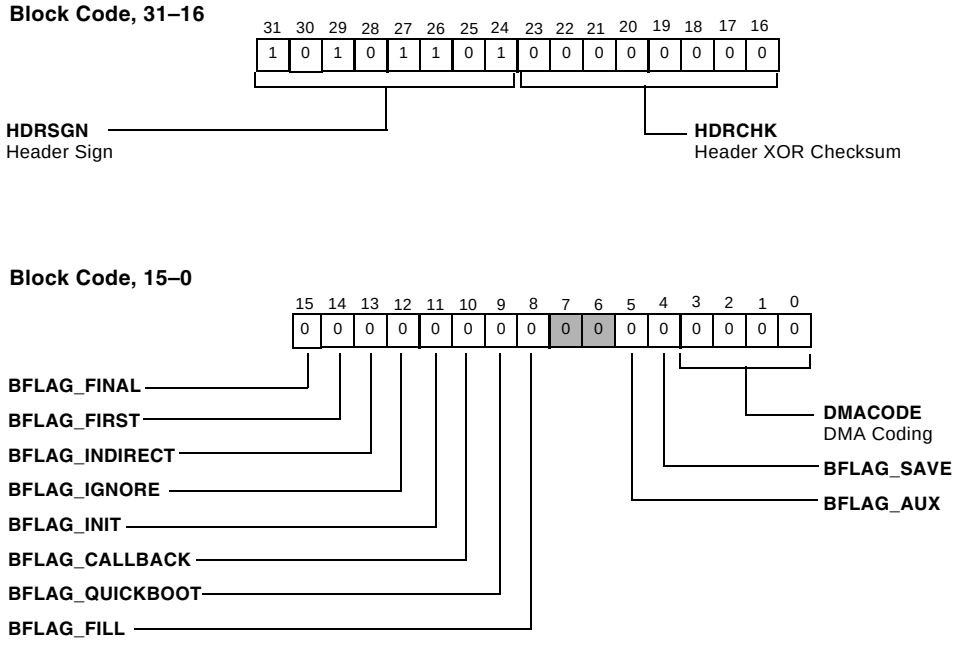

Block Code, 31–16
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
1 0 1 0 1 1 0 1 0 0 0 0 0 0 0 0
HDRSGN
Header Sign
HDRCHK
Header XOR Checksum
Block Code, 15–0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
BFLAG_FINAL
BFLAG_FIRST
BFLAG_INDIRECT
BFLAG_IGNORE
BFLAG_INIT
BFLAG_CALLBACK
BFLAG_QUICKBOOT
BFLAG_FILL
DMACODE
DMA Coding
BFLAG_SAVE
BFLAG_AUX

Figure 17-7. Block Code, 31–0

DMA Code Field

The DMA code (DMACODE) field instructs the boot kernel whether to use 8-bit, 16-bit or 32-bit DMA and how to program the source modifier of a memory DMA. Particularly in case of memory boot modes, this field is interrogated by the boot kernel to differentiate the 8-bit, 16-bit, and 32-bit cases.

The boot kernel tests this field only on the first block and ignores the field in further blocks (See Table 17-3).

Table 17-3. Bus and DMA Width Coding

DMA Code	DMA Width	Source DMA Modify	Application
0	reserved[1]		
1	8-bit	1	Default 8-bit boot from 8-bit source[2]
2	8-bit	2	Zero-padded 8-bit boot from 16-bit EBIU
3	8-bit	4	Zero-padded 8-bit boot from 32-bit EBIU[3]
4	8-bit	8	Zero-padded 8-bit boot from 64-bit EBIU[4]
5	8-bit	16	Zero-padded 8-bit boot from 128-bit EBIU[4]
6	16-bit	2	Default 16-bit boot from 16-bit source[5]
7	16-bit	4	Zero-padded 16-bit boot from 32-bit EBIU[3]
8	16-bit	8	Zero-padded 16-bit boot from 64-bit EBIU[4]
9	16-bit	16	Zero-padded 16-bit boot from 128-bit EBIU[4]
10	32-bit	4	Default 32-bit boot from 32-bit source[3, 5]
11	32-bit	8	Zero-padded 32-bit boot from 64-bit EBIU[4]
12	32-bit	16	Zero-padded 32-bit boot from 128-bit EBIU[4]
13	64-bit	8	Default 64-bit boot from 64-bit source[4]
14	64-bit	16	Zero-padded 64-bit boot from 128-bit EBIU[4]
15	128-bit	16	Default 128-bit boot from 128-bit source[4]

1 Reserved to differentiate from ADSP-BF53x boot streams.
2 Used by all byte-wise serial boot modes.
3 Applicable only to memory boot modes and OTP mode. This code is expected by OTP boot mode.
4 Not supported by ADSP-BF52x Blackfin products.
5 This is the only code supported by NAND flash boot.

Block Flags Field

Table 17-4. Block Flags

Bit	Name	Description
4	BFLAG_SAVE	Saves the memory of this block to off-chip memory in case of power failure or a hibernate request. This flag is not used by the on-chip boot kernel.
5	BFLAG_AUX	Nests special block types as required by special purpose second-stage loaders. This flag is not used by the on-chip boot kernel.
6	Reserved	
7	Reserved	
8	BFLAG_FILL	Tells the boot kernel to not process any payload data. Instead the target memory (specified by the `TARGET ADDRESS` and `BYTE COUNT` fields) is filled with the 32-bit value provided by the `ARGUMENT` word. The fill operation is always performed by 32-bit DMA; therefore target address and byte count must be divisible by four.
9	BFLAG_QUICKBOOT	Processes the block for full boot only. Does not process this block for a quick boot (warm boot).
10	BFLAG_CALLBACK	Calls a subfunction that may reside in on-chip or off-chip ROM or is loaded by an initcode in advance. Often used with the `BFLAG_INDIRECT` switch. If `BFLAG_CALLBACK` is set for any block, an initcode must register the callback function first. The function is called when either the entire block is loaded or the intermediate storage memory is full. The callback function can do advanced processing such as CRC checksum.
11	BFLAG_INIT	This flag causes the boot kernel to issue a `CALL` instruction to the target address of the boot block after the entire block is loaded. The initcode should return by an `RTS` instruction. It may or may not be overwritten by application data later in the boot process. If the code is loaded earlier or resides in ROM, the init block can be zero sized (no payload).

Table 17-4. Block Flags (Continued)

Bit	Name	Description
12	BFLAG_IGNORE	Indicates a block that is not booted into memory. It instructs the boot kernel to skip the number of bytes of the boot stream as specified by BYTE COUNT. In master boot modes, the boot kernel simply modifies its source address pointer. In this case the BYTE COUNT value can be seen as a 32-bit two's-complement offset value to be added to the source address pointer. In slave boot modes, the boot kernel actively loads and changes the payload of the block. In slave modes the byte count must be a positive value.
13	BFLAG_INDIRECT	Boots to an intermediate storage place, allowing for calling an optional callback function, before booting to the destination. This flag is used when the boot source does not have DMA support (TWI for example) and either the destination cannot be accessed by the core (L1 instruction SRAM) or cannot be efficiently accessed by the core (SDRAM or RAM). This flag is also used when CALLBACK requires access to data to calculate a checksum, or when performing tasks such as decryption or decompression.
14	BFLAG_FIRST	This flag, which is only set on the first block of a DXE, tells the boot kernel about the special nature of the TARGET ADDRESS and the ARGUMENT fields. The TARGET ADDRESS field holds the start address of the application. The ARGUMENT field holds the offset to the next DXE.
15	BFLAG_FINAL	This flag causes the boot kernel to pass control over to the application after the final block is processed. This flag is usually set on the last block of a DXE unless multiple DXEs are merged.

The BFLAG_FIRST flag must not be combined with the BFLAG_FILL flag. The BFLAG_FIRST flag may be combined with the BFLAG_IGNORE flag to deposit special user data at the top of the boot stream. Note the special importance of the VisualDSP++ elfloader -readall switch.

Header Checksum Field

The header checksum (HDRCHK) field holds a simple XOR checksum of the other 31 bytes in the boot block header. The boot kernel jumps to the error routine if the result of an XOR operation across all 32 header bytes (including the HDRCHK value) differs from zero. The default error routine is

a simple IDLE; instruction. The user can overwrite the default error handler using the initcode mechanism.

Header Sign Field

The header signature (HDRSGN) byte always reads as 0xAD and is used to verify whether the block pointer actually points to a valid block. The HDRSGN byte can also be used as a boot stream version control. For the ADSP-BF54x, ADSP-BF52x and ADSP-BF51x Blackfin processors, the byte always reads 0xAD. The ADSP-BF53x boot streams always read 0xFF. The ADSP-BF561 boot streams always read 0xA0.

Target Address

This 32-bit field holds the target address where the boot kernel loads the block payload data. When the BFLAG_FILL flag is set, the boot kernel fills the memory with the value stored in the ARGUMENT field starting at this address. If the BFLAG_INIT flag is set the kernel issues a CALL(TARGET ADDRESS) instruction after the optional payload is loaded.

If the BFLAG_FIRST flag is set, the TARGET ADDRESS field contains the start address of the application to which the boot kernel jumps at the end of the boot process. This address will also be stored in the EVT1 register. By default the VisualDSP++ elfloader utility sets this value to 0xFFA0 0000 for compatibility with other Blackfin products.

The target address should be divisible by four, because the boot kernel uses 32-bit DMA for certain operations. The target address must point to valid on-chip or off-chip memory locations. When booting to external memories, the memory controller must first be set up by either the preboot or the initcode mechanism. When booting through peripherals that do not support DMA transfers, such as the TWI or OTP boot mode, the BFLAG_INDIRECT flag must be set if the target address points to L1 instruction memory. For performance reasons this is also recommended when booting to off-chip memories.

For the TWI or OTP boot modes, the VisualDSP++ elfloader utility manages the BFLAG_INDIRECT flag automatically. Refer to the *VisualDSP++ Loader and Utilities* reference guide for manual control of the flag.

Booting to scratchpad memory is not supported. The scratchpad memory functions as a stack for the boot kernel. The L1 data memory locations 0xFF80 7FF0 to 0xFF80 7FFF are used by the boot kernel and should not be overwritten by the application. The memory range used for intermediate storage as controlled by the BFLAG_INDIRECT switch should only be booted after the last BFLAG_INDIRECT bit is processed. By default the address range 0xFF90 7E00–0xFF90 7FFF is used for intermediate storage.

For normal boot operation, the target address points to RAM memory. There are however a few exceptions where the target address can point to on-chip or off-chip ROM. For example a zero-sized BFLAG_INIT block would instruct the boot kernel to call a subroutine residing in ROM or flash memory. This method is used to activate the CRC32 feature.

Byte Count

This 32-bit field tells the boot kernel how many bytes to process. Normally, this is the size of the payload data of a boot block. If the BFLAG_FILL flag is set there is no payload. In this case the BYTE COUNT field uses the value in its ARGUMENT field to tell the boot kernel how many bytes to process.

The byte count is a 32-bit value that should be divisible by four. Zero values are allowed in all block types. Most boot modes are based upon DMA operation which are only 16-bit words for Blackfin processors. The boot kernel may therefore start multiple DMA work units for large boot blocks. This enables a single block to fill to zero the entire SDRAM memory, for example, resulting in compact boot streams. The HWAIT signal may toggle for each work unit.

If the BFLAG_IGNORE flag is set, the byte count is used to redirect the boot source pointer to another memory location. In master boot modes, the byte count is a two's-complement (signed long integer) value. In slave boot modes, the value must be positive.

Argument

This 32-bit field is a user variable for most block types. The value is accessible by the initcode or the callback routine and can therefore be used for optional instructions to these routines. When the CRC32 feature is activated, the ARGUMENT field holds the checksum over the payload of the block.

When the BFLAG_FILL flag is set there is no payload. The argument contains the 32-bit fill value, which is most likely a zero.

If the BFLAG_FIRST flag is set, the argument contains the relative next-DXE pointer for multi-DXE applications. For single-DXE applications the field points to the next free boot source address after the current DXE's boot stream.

Boot Host Wait (HWAIT) Feedback Strobe

The HWAIT feedback strobe is a handshake signal that is used to hold off the host device from sending further data while the boot kernel is busy.

On ADSP-BF52x processors this feature is implemented by a GPIO that is toggled by the boot kernel as required. The PG0 GPIO is used for this purpose.

The signal polarity of the HWAIT strobe is programmable by an external resistor in the 10 kΩ range.

A pull-up resistor instructs the HWAIT signal to be active high. In this case the host is permitted to send header and footer data when HWAIT is low,

but should pause while HWAIT is high. This is the mode used in SPI slave boot on other Blackfin products.

Similarly, a pull-down resistor programs active-low behavior.

Note that the HWAIT signal is implemented slightly differently than on ADSP-BF53x Blackfin processors. In the ADSP-BF52x processors, the meaning of the pulling resistor is inverted and HWAIT is asserted by default during reset and preboot.

After preboot, the boot kernel first senses the polarity on the respective HWAIT pin. Then it enables the output driver but keeps the signal in its asserted state. The signal is not released until the boot kernel is ready for data, or when a receive DMA is started. As soon as the DMA completes, HWAIT becomes active again.

The boot host wait signal holds the host from booting in any slave boot mode and prevents it from being overrun with data. The HWAIT signal is, however, available in all boot modes with the exception of the NAND flash boot mode.

In general the host device must interrogate the HWAIT signal before every word that is sent. This requirement can be relaxed for boot modes using on-chip peripherals that feature larger receive FIFOs. However, the host must not rely on the DMA FIFO since its content is cleared at the end of a DMA work unit.

While the HWAIT signal is only used for boot purposes, it may also play a significant role after booting. In slave boot modes, for example, the host device does not necessarily know whether the Blackfin processor is in an active mode or a power-down mode. For example, the HWAIT signal can be used to signal when the processor is in hibernate mode.

Using HWAIT as Reset Indicator

While the HWAIT signal is mandatory in some boot modes, it is optional in others. When not required for booting, the behavior of the HWAIT signal

can be changed by programming the OTP_RESETOUT_HWAIT bit in OTP page PBS00L.

If this bit is set, HWAIT does not toggle during the boot process. Rather, after page PBS00L is processed (and therefore the PLL has settled) the pre-boot routine first enables the HWAIT GPIO as an input and senses its state. Then HWAIT becomes an output and is driven to the invert of the state that is sensed. An external pulling resistor is required. If using a pull-up resistor, the HWAIT signal is driven low for the rest of the boot process (and beyond). If using a pull-down resistor, HWAIT is driven high.

With a pull-down resistor, this feature can be used to simulate an active-low reset output. When the processor is reset, or in hibernate, the GPIO is in a high impedance state and HWAIT is pulled low by the resistor. As soon as the processor recovers and has settled the PLL again, the HWAIT is driven high and can alert external circuits.

Boot Termination

After the successful download of the application into the bootable memory, the boot kernel passes control to the user application. By default this is performed by jumping to the vector stored in the EVT1 register. The boot kernel provides options to execute an RTS instruction or a RAISE 1 instruction instead. The default behavior can be changed by an initcode routine. The EVT1 register is updated by the boot kernel when processing the BFLAG_FIRST block. See "Servicing Reset Interrupts" on page 17-10 to learn how the application can take control.

Before the boot kernel passes program control to the application it does some housekeeping. Most of the registers that were used are changed back to their default state but some register values may differ for individual boot modes. DMA configuration registers and primary register control registers (UART0_LCR, SPI_CTL, HOST_CONTROL, etc.) are restored, while others are purposely not restored. For example SPI_BAUD, UART0_DLH and

UART_DLL remain unchanged so that settings obtained during the booting process are not lost.

Single Block Boot Streams

The simplest boot stream consists of a single block header and one contiguous block of instructions. With appropriate flag instructions the boot kernel loads the block to the target address and immediately terminates by executing the loaded block.

Table 17-5 shows an example of a single block boot stream header that could be loaded from any serial boot mode. It places a 256-byte block of instructions at L1 instruction SRAM address 0xFFA0 0000. The flags BFLAG_FIRST and BFLAG_FINAL are both set at the same time. Advanced flags, such as BFLAG_IGNORE, BFLAG_INIT, BFLAG_CALLBACK and BFLAG_FILL, do not make sense in this context and should not be used.

Table 17-5. Header for a Single Block Boot Stream

Field	Value	Comments
BLOCK CODE	0xAD33 C001	0xAD00 0000 | XORSUM | BFLAG_FINAL | BFLAG_FIRST | (DMACODE & 0x1)
TARGET ADDRESS	0xFFA0 0000	Start address of block and application code
BYTE COUNT	0x0000 0100	256 bytes of code
ARGUMENT	0x0000 0100	Functions as next-DXE pointer in multi-DXE boot streams

With the BFLAG_FIRST flag set, the ARGUMENT field functions as the next-DXE pointer. This is a relative pointer to the next free source address or to the next DXE start address in a multi-DXE stream.

Direct Code Execution

Applications may want to avoid long booting times and start code execution directly from 16-bit flash or SDRAM memory. This feature is called

direct code execution. This is a special case of boot termination that replaces the no-boot/bypass mode in the ADSP-BF53x Blackfin processors.

An initial boot block header is needed for the processor to fetch and execute program code from the boot device as early as possible. The safety mechanisms of the block, such as the header signature and the XOR checksum, avoid unpredictable processor behavior due to the boot memory not being programmed with valid data yet. Rather than blindly executing code, the boot kernel first executes the preboot routine for system customization, then loads the first block header and checks it for consistency. If the block header is corrupted, the boot kernel goes into a safe idle state and does not start code execution.

If the initial block header checks good, the boot kernel interrogates the block flags. If the block has the BFLAG_FINAL flag set, the boot kernel immediately terminates and jumps directly to the address stored in the EVT1 register. To cause the boot kernel to customize the EVT1 register in advance, the initial blocks must also have the BFLAG_FIRST flag set. The TARGET ADDRESS field is then copied to the EVT1 register. In this way, the TARGET ADDRESS field of the initial block defines the start address of the application.

For example in BMODE = 0001, when the block header described in Table 17-6 on page 17-36 is placed at address 0x2000 0000, the boot kernel is instructed to issue a JUMP command to address 0x2000 0020.

The development tools must be instructed to link the above block to address 0x2000 0000 and the application code to address 0x2000 0020.

An example shown in "Direct Code Execution" on page 17-148 illustrates how this is accomplished using the VisualDSP++ tools suite.

Table 17-6. Initial Header for Direct Code Execution in BMODE = 0001

Field	Value	Comments
BLOCK CODE	0xAD7B D006	0xAD00 0000 | XORSUM | BFLAG_FINAL | BFLAG_FIRST | BFLAG_IGNORE | (DMACODE & 0x6)
TARGET ADDRESS	0x2000 0020	Start address of application code
BYTE COUNT	0x0000 0010	Ignores 16 bytes to provide space for control data such as version code and build data. This is optional and can be zero.
ARGUMENT	0x0000 0010	Functions as next-DXE pointer in multi-DXE boot streams

Similarly for direct code execution in the SDRAM boot mode (BMODE = 1010), an initial block as shown in Table 17-7 has to be linked to address 0x0000 0010.

Table 17-7. Initial Header for Direct Code Execution in BMODE = 1010

Field	Value	Comments
BLOCK CODE	0xAD5B D006	0xAD000000 | XORSUM | BFLAG_FINAL | BFLAG_FIRST | BFLAG_IGNORE | (DMACODE & 0x6)
TARGET ADDRESS	0x0000 0020	Start address of application code
BYTE COUNT	0x0000 0000	No bubble for control data
ARGUMENT	0x0000 0000	Functions as next-DXE pointer in multi-DXE boot streams

For multi-DXE boot streams, Figure 17-11 on page 17-56 shows a linked list of initial blocks that represent different applications.

Advanced Boot Techniques

Initialization Code

Initcode routines are subroutines that the boot kernel calls during the booting process. The user can customize and speed up the booting mechanisms using this feature. Traditionally, an initcode is used to set up system PLL, bit rates, wait states and the SDRAM controller. If executed early in the boot process, the boot time can be significantly reduced.

After the payload data is loaded for a specific boot block, if the `BFLAG_INIT` flag is set, the boot kernel issues a `CALL` instruction to the target address of the block.

On ADSP-BF52x Blackfin processors, initcode routines follow the C language calling convention so they can be coded in C language or assembly.

The expected prototype is

```
void initcode(ADI_BOOT_DATA* pBootStruct);
```

The VisualDSP++ header files define the `ADI_BOOT_INITCODE_FUNC` type:

```
typedef void ADI_BOOT_INITCODE_FUNC (ADI_BOOT_DATA* ) ;
```

Optionally, the initcode routine can interrogate the formatting structure and customize its own behavior or even manipulate the regular boot process. A pointer to the structure is passed in the `R0` register. Assembly coders must ensure that the routine returns to the boot kernel by a terminating `RTS` instruction.

Initcodes can rely on the validity of the stack, which resides in scratchpad memory. The `ADI_BOOT_DATA` structure resides on the stack. Rules for register usage conform to the compiler conventions. See the *VisualDSP++ C/C++ Compiler and Library Manual* for more information.

In the simple case, initcodes consist of a single instruction section and are represented by a single block within the boot stream. This block has the BFLAG_INIT bit set.

An init block can consist of multiple sections where multiple boot blocks represent the initcode within the boot stream. Only the last block has the BFLAG_INIT bit set.

The VisualDSP++ elfloader utility ensures that the last of these blocks vectors to the initcode entry address. The utility instructs the on-chip boot ROM to execute a CALL instruction to the given target address.

When the on-chip boot ROM detects a block with the BFLAG_INIT bit set, it boots the block into Blackfin memory and then executes it by issuing a CALL to its target address. For this reason, every initcode must be terminated by an RTS instruction to ensure that the processor vectors back to the on-chip boot ROM for the rest of the boot process.

Sometimes initcode boot blocks have no payload and the BYTE COUNT field is set to zero. Then the only purpose of the block may be to instruct the boot kernel to issue the CALL instruction.

Initcode routines can be very different in nature. They might reside in ROM or SRAM. They might be called once during the booting process or multiple times. They might be volatile and be overwritten by other boot blocks after executing, or they might be permanently available after boot time. The boot kernel has no knowledge of the nature of initcodes and has no restrictions in this regard. Refer to the *VisualDSP++ Loader and Utilities Manual* for how this feature is supported by the tools chain.

It is the user's responsibility to ensure that all code and data sections that are required by the initcode are present in memory by the time the initcode executes. Special attention is required if initcodes are written in C or C++ language. Ensure that the initcode does not contain calls to the runtime libraries. Do not assume that parts of the runtime environment, such as the heap are fully functional. Ensure that all runtime components are loaded and initialized before the initcode executes.

The VisualDSP++ elfloader utility provides two different mechanisms to support the initcode feature.

- The `-init initcode.dxe` command line switch
- The `-initcall address/symbol` command line switch

If enabled by the VisualDSP++ elfloader `-init initcode.DXE` command line switch, the initcode is added to the beginning of the boot stream. Here, `initcode.DXE` refers to the user-provided custom initialization executable— a separate VisualDSP++ project. Figure 17-8 on page 17-40 shows a boot stream example that performs the following steps.

1. Boot initcode into L1 memory.
2. Execute initcode.
3. Initcode initializes the SDRAM controller and returns.
4. Overwrite initcode with final application code.
5. Boot data/code into SDRAM.
6. Continue program execution with block n.

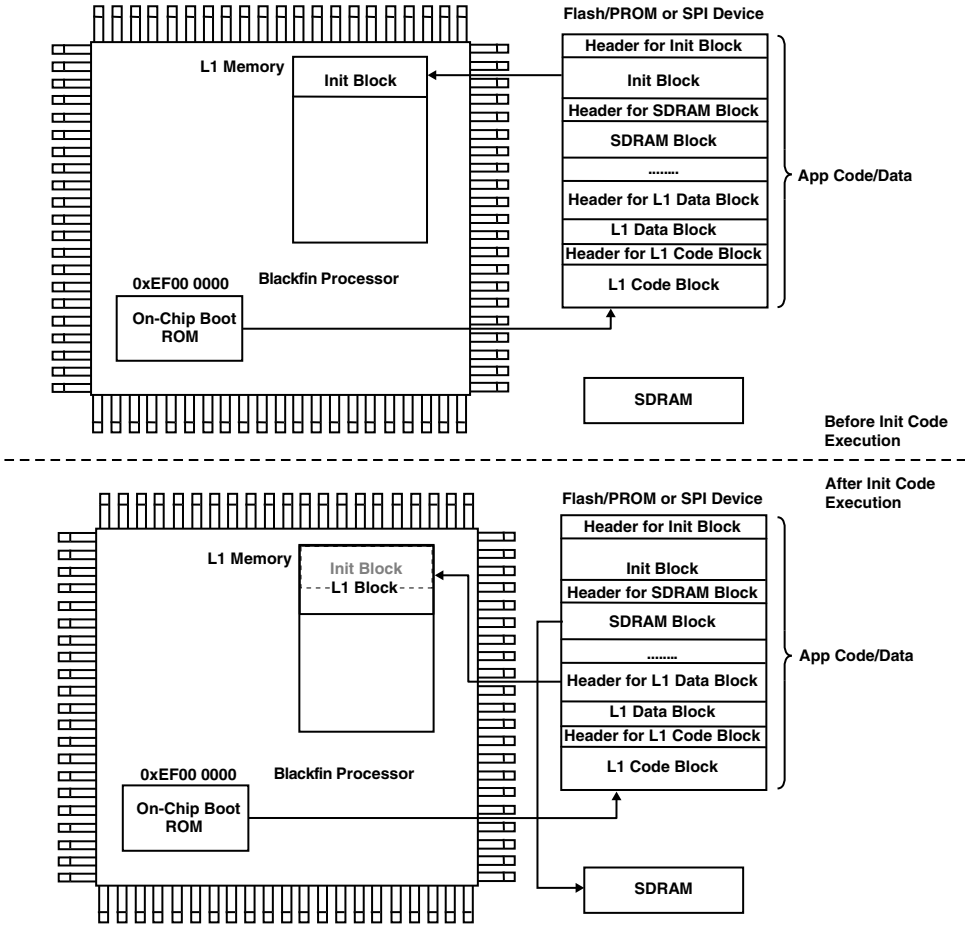


Figure 17-8. Initialization Code Execution/Boot

Although `initcode.DXE` files are built as VisualDSP++ projects, they differ from standard projects. Initcodes provide only a callable sub-function, so they look more like a library than an application. Nevertheless, unlike

library files (.DLB file extension), the symbol addresses have already been resolved by the linker.

An initcode is always a heading for the regular application code. Consequently whether the initcode consists of one or multiple blocks, it is not terminated by a BFLAG_FINAL bit indicator—this would cause the boot ROM to terminate the boot process.

It is advantageous to have a clear separation between the initcode and the application by using the -init switch. If this separation is not needed, the elfloader -initcall command-line switch might be preferred. It enables fractions of the application code to be traded as initcode during the boot process. See the *VisualDSP++ Loader and Utilities Manual* for further details.

Initcode examples are shown in "Programming Examples" on page 17-138.

Quick Boot

In some booting scenarios, not all memories need to be re-initialized. For example in a wake-up from hibernate state, off-chip SRAM might not be impacted if it was powered while the processor was in hibernate state. Dynamic RAM might also not be impacted if it was put into self-refresh mode before the processor powered down.

The ADSP-BF52x processor's boot kernel can conditionally process boot blocks. The normal scenario is all boot, the shortened version is quick boot. It relies on the following primitives.

- The SYSCR register is read to determine what kind of boot is expected from the boot kernel. Refer to Figure 17-39 on page 17-102.

  The WURESET bit is used to distinguish between cold boot and warm boot situations and to identify wake-up from hibernate situations.

The BCODE bit field in the SYSCR register can overrule the native decision of the boot kernel for a software boot. See the flowchart in Figure 17-1 on page 17-9.

- The BFLAG_WAKEUP bit in the dFlag word of the ADI_BOOT_DATA structure indicates that the final decision was to perform a quick boot. If the boot kernel is called from the application, then the application can control the boot kernel behavior by setting the BFLAG_WAKEUP flag accordingly. See the dFlags variable on Figure 17-52 on page 17-120.
- The BFLAG_QUICKBOOT flag in the BLOCK CODE word of the block header controls whether the current block is ignored for quick boot.

If both the global BFLAG_WAKEUP and the block-specific BFLAG_QUICKBOOT flags are set, the boot kernel ignores those blocks. But since the BFLAG_INIT, BFLAG_CALLBACK, BFLAG_FINAL, and BFLAG_AUX flags are internally cleared and the BFLAG_IGNORE flag is toggled, through double negation, the "ignore the ignore block" command instructs the boot kernel to process the block.

Although the BFLAG_INIT flag is suppressed in quick boot, the user may not want to combine the BFLAG_INIT flag with the BFLAG_QUICKBOOT flag. The initialization code can interrogate the BFLAG_WAKEUP flag and execute conditional instructions. For more information see "Quickboot With Restore From SDRAM" on page 17-145.

Indirect Booting

The processor's boot kernel provides a control mechanism to let blocks either boot directly to their final destination or load to an intermediate

storage place, then copy the data to the final destination in a second step. This feature is motivated by the following requirements.

- Some boot modes such as TWI do not use DMA. They load data by core instruction. The core cannot access some memories directly (for example L1 instruction SRAM), or is less efficient than the DMA in accessing some memories (for example, external SDRAM).
- In some advanced booting scenarios, the core needs to access the boot data during the booting process, for example in processing decompression, decryption and checksum algorithms at boot time. The indirect booting option helps speed-up and simplify such scenarios. Software accesses off-chip memory less efficiently and cannot access data directly if it resides in L1 instruction SRAM.

Indirect booting is not a global setting. Every boot block can control its own processing by the `BFLAG_INDIRECT` flag in the block header.

In general a boot block may not fit into the temporary storage memory so the boot kernel processes the block in multiple steps. The larger the temporary buffer, the faster the boot process. By default the L1 data memory region between 0xFF90 7E00 and 0xFF90 7FFF is used for intermediate storage. Initialization code can alter this region by modifying the `pTempBuffer` and `dTempByteCount` variables in the `ADI_BOOT_DATA` structure. The default region is at the upper end of a physical memory block. When increasing the `dTempByteCount` value, `pTempBuffer` also has to change.

Callback Routines

Callback routines, like initialization codes, are user-defined subroutines called by the boot kernel at boot time. The `BFLAG_CALLBACK` flag in the block header controls whether the callback routine is called for a specific block.

There are several differences between initcodes and callback routines. While the BFLAG_INIT flag causes the boot kernel to issue a CALL instruction to the target address of the specific boot block, the BFLAG_CALLBACK flag causes the boot kernel to issue a CALL instruction to the address held by the pCallBackFunction pointer in the ADI_BOOT_DATA structure. While a boot stream can have multiple individual initcodes, it can have just one callback routine. In the standard boot scenario, the callback routine has to be registered by an initcode prior to the first block that has the BFLAG_CALLBACK flag set.

The purpose of the callback routine is to apply standard processing to the block data. Typically, callback routines contain checksum, decryption, decompression, or hash algorithms. Checksum or hash words can be passed through the block header ARGUMENT field.

Since callback routines require access to the payload data of the boot blocks, the block data must be loaded before it can be processed. Unlike initcodes, a callback usually resides permanently in memory. If the block is loaded to L1 instruction memory or off-chip memory, the BFLAG_CALLBACK flag is likely combined with the BFLAG_INDIRECT bit. The boot kernel performs these steps in the following order.

1. Data is loaded into the temporary buffer defined by the pTempBuffer variable.
2. The CALL to the pCallBackFunction is issued.
3. After the callback routine returns, the memory DMA copies data to the destination.

If a block does not fit into the temporary buffer, for example when the BLOCK COUNT is greater than the dTempByteCount variable, the three steps are executed multiple times until all payload data is loaded and processed. The boot kernel passes the parameter dCbFlags to the callback routine to tell it that it is being invoked the first or the last time for a specific block. To store intermediate results across multiple calls the callback routine can

use the uwUserShort and dUserLong variables in the ADI_BOOT_DATA structure.

Callback routines meet C language calling conventions for subroutines. The prototype is as follows.

```
s32 CallBackFunction (ADI_BOOT_DATA* pBootStruct,
ADI_BOOT_BUFFER* pCallbackStruct, s32 dCbFlags);
```

The VisualDSP++ header file defines the ADI_BOOT_CALLBACK_FUNC type the following way:

```
typedef s32 ADI_BOOT_CALLBACK_FUNC (ADI_BOOT_DATA*,
ADI_BOOT_BUFFER*, s32 ) ;
```

The pBootStruct argument is passed in R0 and points to the ADI_BOOT_DATA structure used by the boot kernel. These are handled by the pTempBuffer and dTempByteCount variables as well as the pHeader pointer to the ARGUMENT field. The callback routine may process the block further by modifying the pTempBuffer and dTempByteCount variables.

The pCallbackStruct structure passed in R1 provides the address and length of the data buffer. When the BFLAG_INDIRECT flag is not set, the pCallbackStruct contains the target address and byte count of the boot block. If the BFLAG_INDIRECT flag is set, the pCallbackStruct contains a copy of the pTempBuffer. Depending on the size of the boot block and processing progress, the byte count provided by pCallbackStruct equals either dTempByteCount or the remainder of the byte count.

When the BFLAG_INDIRECT flag is set along with the BFLAG_CALLBACK flag, memory DMA is invoked by the boot kernel after the callback routine returns. This memory DMA relies on the pCallbackStruct structure not the global pTempBuffer and dTempByteCount variables.

The callback routine can control the source of the memory DMA by altering the content of the pCallbackStruct structure, as may be required if the callback routine performs data manipulation such as decompression.

The dCbFlags parameter passed in R2 tells the callback routine whether it is invoked the first time (CBFLAG_FIRST) or whether it is called the last time (CBFLAG_FINAL) for a specific block. The CBFLAG_DIRECT flag indicates that the BFLAG_INDIRECT bit is not active so that the callback routine will only be called once per block. When the CBFLAG_DIRECT flag is set, the CBFLAG_FIRST and CBFLAG_FINAL flags are also set.

```
#define CBFLAG_FINAL        0x0008
#define CBFLAG_FIRST        0x0004
#define CBFLAG_DIRECT       0x0001
```

A callback routine also has a boolean return parameter in register R0. If the return value is non-zero, the subsequent memory DMA does not execute. When the CBFLAG_DIRECT flag is set, the return value has no effect.

Error Handler

While the default handler simply puts the processor into idle mode, an initcode routine can overwrite this pointer to create a customized error handler. The expected prototype is

```
void ErrorFunction (ADI_BOOT_DATA* pBootStruct, void
*pFailingAddress);
```

Use an initcode to write the entry address of the error routine to the pErrorFunction pointer in the ADI_BOOT_DATA structure. The error handler has access to the boot structure and receives the instruction address that triggered the error.

CRC Checksum Calculation

The ADSP-BF52x Blackfin processors provide an initcode and a callback routine in ROM that can be used for CRC32 checksum generation during boot time. The checksum routine only verifies the payload data of the blocks. The block headers are already protected by the native XOR checksum mechanism.

Before boot blocks can be tagged with the BFLAG_CALLBACK flag to enable checksum calculation on the blocks, the boot stream must contain an initcode block with no payload data and with the CRC32 polynomial in the block header ARGUMENT word.

The initcode registers a proper CRC32 wrapper to the pCallBackFunction pointer. The registration principle is similar to the XOR checksum example shown in "Programming Examples" on page 17-138.

Load Functions

With the exception of the Host DMA boot modes, all boot modes are processed by a common boot kernel algorithm. The major customization is done by a subroutine that must be registered to the pLoadFunction pointer in the ADI_BOOT_DATA structure. Its simple prototype is as follows.

```
void LoadFunction (ADI_BOOT_DATA* pBootStruct);
```

The VisualDSP++ header files define the following type:

```
typedef void ADI_BOOT_LOAD_FUNC (ADI_BOOT_DATA* ) ;
```

For a few scenarios some of the flags in the dFlags word of the ADI_BOOT_DATA structure, such as BFLAG_PERIPHERAL and BFLAG_SLAVE, slightly modify the boot kernel algorithm.

The boot ROM contains several load functions. One performs a memory DMA for flash boot, others perform peripheral DMAs or load data from booting source by polling operation. The first is reused for fill operation and indirect booting as well.

In second-stage boot schemes, the user can create customized load functions or reuse the original BFROM_PDMA routine and modify the pDmaControlRegister, pControlRegister and dControlValue values in the ADI_BOOT_DATA structure. The pDmaControlRegister points to the DMAx_CONFIG or MDMA_Dx_CONFIG register. When the BFLAG_SLAVE flag is not set, the pControlRegister and dControlValue variables instruct the

peripheral DMA routine to write the control value to the control register every time the DMA is started.

Load functions written by users must meet the following requirements.

- Protect against `dByteCount` values of zero.
- Multiple DMA work units are required if the `dByteCount` value is greater than 65536.
- The `pSource` and `pDestination` pointers must be properly updated.

In slave boot modes, the boot kernel uses the address of the `dArgument` field in the `pHeader` block as the destination for the required dummy DMAs when payload data is consumed from `BFLAG_IGNORE` blocks. If the load function requires access to the block's `ARGUMENT` word, it should be read early in the function.

The most useful load functions `BFROM_MDMA` and `BFROM_PDMA` are accessible through the jump table. Others, do not have entries in the jump table. Their start address can be determined with the help of the hook routine when calling the respective `BFROM_SPIBOOT`, `BFROM_OTPBOOT` etc. functions. In this way they can be repurposed for runtime utilization.

Calling the Boot Kernel at Runtime

The boot kernel's primary purpose is to boot data to memory after power-up and reset cycles. However some of the routines used by the boot kernel might be of general value to the application. The boot ROM supports reuse of these routines as C-callable subroutines. Programs such as second-stage boot kernels, boot managers, and firmware update tools may call the function in the ROM at runtime. This could load entirely different applications or a fraction of an application, such as a code overlay or a coefficient array.

To call these boot kernel subroutines, the boot ROM provides an API at address 0xEF00 0000 in the form of a jump table.

When calling functions in the boot ROM, the user must ensure the presence of a valid stack following C language conventions. See the VisualDSP++ Compiler documentation for details.

Debugging the Boot Process

If the boot process fails, very little information can be gained by watching the chip from outside. In master boot modes, the interface signals can be observed. In slave boot modes only the HWAIT signal tells about the progress of the boot process.

However, by using the emulator, there are many possibilities for debugging the boot process. The entire source code of the boot kernel is provided with the VisualDSP++ installation. This includes the project executable (DXE) file. The LOAD SYMBOLS feature of the VisualDSP++ IDDE helps to navigate the program. Note that the content of the ROM might differ between silicon revisions. Hardware breakpoints and single-stepping capabilities are also available.

Table 17-8 shows program symbols that are of interest.

Table 17-8. Boot Kernel Symbols for Debug

Symbol	Comment
_bootrom.assert.default	If the program counter halts at the IDLE instruction at the _bootrom.assert.default address, either the boot kernel or the preboot has detected an error condition and will not continue the boot process. A misformatted boot stream, or invalid PBS settings are the most likely causes of such an error. The RETS register points to the failing routine. When stepping a couple of instructions further, there is a way to ignore the error and to continue the boot process by clearing the >ASTAT register while the emulator steps over the subsequent IF CC JUMP 0 instruction.
_bootrom.bootmenu	If the emulator hits a hardware breakpoint at the _bootrom.bootmenu address, this indicates that the preboot returned properly. Otherwise the program may hang during preboot due to improper PBS settings or invalid boot modes.

Table 17-8. Boot Kernel Symbols for Debug (Continued)

Symbol	Comment
_bootrom.bootkernel.entry	If the emulator hits a hardware breakpoint at the `_bootrom.bootkernel.entry` label, this indicates that device detection or autobaud returned properly.
_bootrom.bootkernel.breakpoint	This is a good address to place a hardware breakpoint. The boot kernel loads a new block header at this breakpoint. The block header can be watched at address 0xFF80 7FF0 or wherever the `pHeader` points to.
_bootrom.bootkernel.initcode	All payload data of the current block is loaded by the time the program passes the `_bootrom.bootkernel.initcode` label. The boot kernel is about to interrogate the `BFLAG_INIT` flag. If set, the initcode can be debugged.
_bootrom.bootkernel.exit	Once the boot kernel arrives at the `_bootrom.bootkernel` exit label, it detects a `BFLAG_FINAL` flag. After some housekeeping, it jumps to the `EVT1` vector.

The boot kernel also generates a circular log file in scratch pad memory. While the `pLogBuffer` and the `dLogByteCount` variables describe the location and dimension of the log buffer, the `pLogCurrent` points to the next free location in the buffer. The log file is updated whenever the kernel passes the `_bootrom.bootkernel.breakpoint` label.

At each pass, nine 32-bit words are written to the log file, as follows.

- block code word (`dBlockCode`) of the block header
- target address (`pTargetAddress`) of the block header
- byte count (`dByteCount`) of the block header
- argument word (`dArgument`) of the block header
- source pointer (`pSource`) of the boot stream
- block count (`dBlockCount`)
- internal copy of the `dBlockCode` word OR'ed with `dFlags`
- content of the `SEQSTAT` register
- 0xFFFF FFFA (-6) constant

The ninth word is overwritten by the next entry set, so that 0xFFFF FFFA always marks the last entry in the log file.

Most of the data structures used by the boot kernel reside on the stack in scratchpad memory. While executing the boot kernel routine (excluding subroutines), the `P5` points to the `ADI_BOOT_DATA` structure. Type "`(ADI_BOOT_DATA*) $P5`" in the VisualDSP++ expression window to see the structure content.

Boot Management

Blackfin processor hardware platforms may be required to run different software at different times. An example might be a system with at least one application and one in-the-field firmware upgrade utility. Other systems may have multiple applications, one starting then terminating, to be replaced by another application. Conditional booting is called boot management. Some applications may self-manage their booting rules, while

others may have a separate application that controls the process, namely a boot manager.

In a master boot mode where the on-chip boot kernel loads the boot stream from memory, the boot manager is a piece of Blackfin software which decides at runtime what application is booted next. This may simply be based on the state of a GPIO input pin interrogated by the boot manager, or it may be the conclusion of complex system behavior.

Slave boot scenarios are different from master boot scenarios. In slave boot modes, the host masters boot management by setting the Blackfin processor to reset and then applying alternate boot data. Optionally, the host could alter the BMODE configuration pins, resulting in little impact to the Blackfin processor since the intelligence is provided by the host device.

Booting a Different Application

The boot ROM provides a set of user-callable functions that help to boot a new application (or a fraction of an application). Usually there is no need for the boot manager to deal with the format details of the boot stream.

These functions are:

- BFROM_MEMBOOT discussed in "Flash Boot Modes" on page 17-59 and "SDRAM Boot Mode" on page 17-63
- BFROM_TWIBOOT discussed in "TWI Master Boot Mode" on page 17-72
- BFROM_SPIBOOT discussed in "SPI Master Boot Modes" on page 17-65
- BFROM_OTPBOOT discussed in "OTP Boot Mode" on page 17-80
- BFROM_NANDBOOT discussed in "NAND Flash Boot Mode" on page 17-86

The user application, the boot manager application, or an initcode can call these functions to load the requested boot data. Using the BFLAG_RETURN flag the user can control whether the routine simply returns to the calling function or executes the loaded application immediately.

These ROM functions expect the start address of the requested boot stream as an argument. For BFROM_MEMBOOT, this is a Blackfin memory address, for BFROM_TWIBOOT and BFROM_SPIBOOT it is a serial address. The SPI function can also accept the code for the GPIO pin that controls the device select strobe of the SPI memory.

Multi-DXE Boot Streams

If the start addresses of all the boot streams are predefined, the boot manager needs only to call the ROM functions directly. However since the addresses tend to vary from build to build they may have to be calculated at runtime.

In the world of the VisualDSP++ elfloader, a boot stream is always generated from a DXE file. It is therefore common to talk about multi-DXE or multi-application booting. When the elfloader utility accepts multiple DXE files on its command line, it generates a contiguous boot image by default. The second boot stream is appended immediately to the first one. Since the utility updates the ARGUMENT field of all BFLAG_FIRST blocks, the ARGUMENT field of a BFLAG_FIRST block is called next-DXE pointer (NDP).

The next-DXE pointer of the first DXE boot stream points relatively to the start address of the second DXE boot stream. A multi-DXE boot image can be seen as a linked list of boot streams. The next-DXE pointer of the last DXE boot stream points relatively to the next free address. This is illustrated by an example shown in the next two figures. Figure 17-9 on page 17-54 shows a commented sketch as an example. Figure 17-10 on page 17-55 shows a screenshot of the Blackfin loader file viewer utility for the same example. The LdrViewer utility is not part of the VisualDSP++ tools suite. It is a third-party freeware product available on www.dolomitics.com.

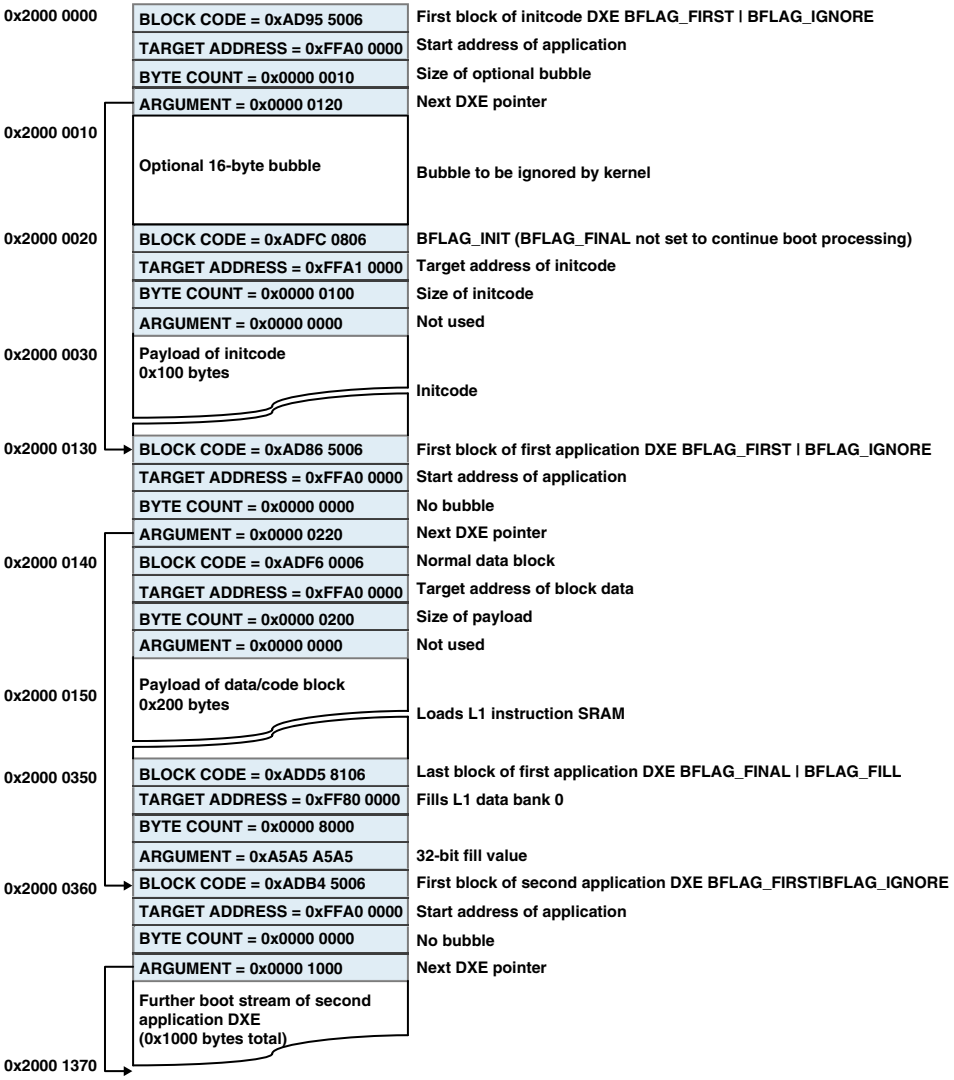

0x2000 0000
BLOCK CODE = 0xAD95 5006
First block of initcode DXE BFLAG_FIRST | BFLAG_IGNORE
TARGET ADDRESS = 0xFFA0 0000
Start address of application
BYTE COUNT = 0x0000 0010
Size of optional bubble
ARGUMENT = 0x0000 0120
Next DXE pointer
0x2000 0010
Optional 16-byte bubble
Bubble to be ignored by kernel
0x2000 0020
BLOCK CODE = 0xADFC 0806
BFLAG_INIT (BFLAG_FINAL not set to continue boot processing)
TARGET ADDRESS = 0xFFA1 0000
Target address of initcode
BYTE COUNT = 0x0000 0100
Size of initcode
ARGUMENT = 0x0000 0000
Not used
0x2000 0030
Payload of initcode
0x100 bytes
Initcode
0x2000 0130
BLOCK CODE = 0xAD86 5006
First block of first application DXE BFLAG_FIRST | BFLAG_IGNORE
TARGET ADDRESS = 0xFFA0 0000
Start address of application
BYTE COUNT = 0x0000 0000
No bubble
ARGUMENT = 0x0000 0220
Next DXE pointer
0x2000 0140
BLOCK CODE = 0xADF6 0006
Normal data block
TARGET ADDRESS = 0xFFA0 0000
Target address of block data
BYTE COUNT = 0x0000 0200
Size of payload
ARGUMENT = 0x0000 0000
Not used
0x2000 0150
Payload of data/code block
0x200 bytes
Loads L1 instruction SRAM
0x2000 0350
BLOCK CODE = 0xADD5 8106
Last block of first application DXE BFLAG_FINAL | BFLAG_FILL
TARGET ADDRESS = 0xFF80 0000
Fills L1 data bank 0
BYTE COUNT = 0x0000 8000
ARGUMENT = 0xA5A5 A5A5
32-bit fill value
0x2000 0360
BLOCK CODE = 0xADB4 5006
First block of second application DXE BFLAG_FIRST|BFLAG_IGNORE
TARGET ADDRESS = 0xFFA0 0000
Start address of application
BYTE COUNT = 0x0000 0000
No bubble
ARGUMENT = 0x0000 1000
Next DXE pointer
Further boot stream of second
application DXE
(0x1000 bytes total)
0x2000 1370

Figure 17-9. Multi-DXE Boot Stream Example for Flash Boot

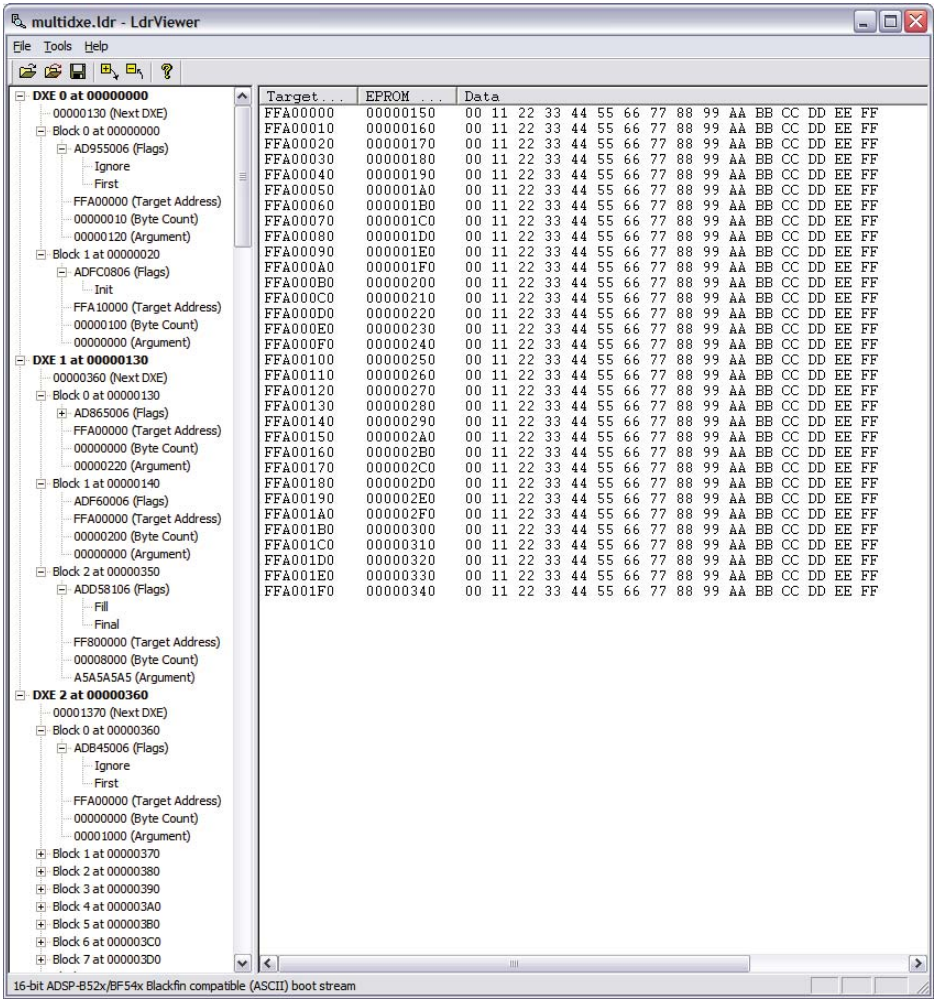


Figure 17-10. LdrViewer Screen Shot

Boot management principles are not only applicable to multi-DXE boot streams. The same scheme, as shown in Figure 17-11 on page 17-56, can be applied to direct code executions of multiple applications. See "Direct Code Execution" on page 17-34 for more information. The example shows a linked list of initial block headers that instruct the boot kernel to

terminate immediately and to start code execution at the address provided by the TARGET ADDRESS field of the individual blocks. There is nothing in the boot ROM that prevents multi-DXE applications from mixing regular boot streams and direct code execution blocks.

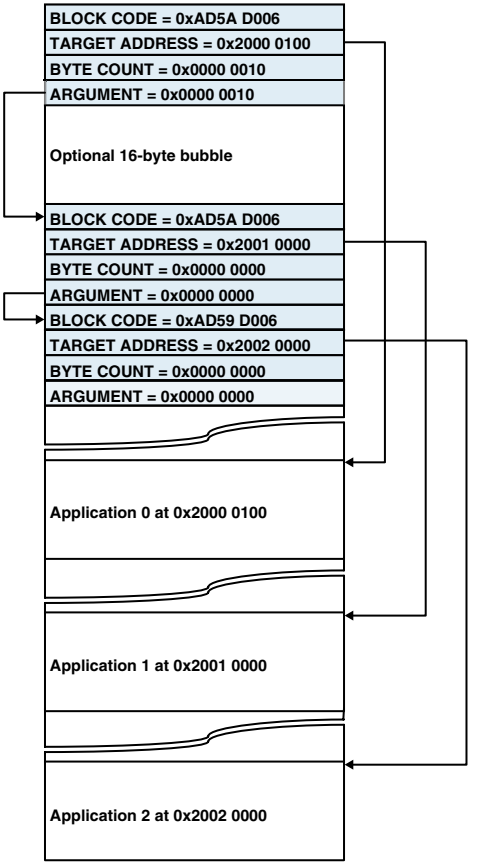

BLOCK CODE = 0xAD5A D006
TARGET ADDRESS = 0x2000 0100
BYTE COUNT = 0x0000 0010
ARGUMENT = 0x0000 0010
Optional 16-byte bubble
BLOCK CODE = 0xAD5A D006
TARGET ADDRESS = 0x2001 0000
BYTE COUNT = 0x0000 0000
ARGUMENT = 0x0000 0000
BLOCK CODE = 0xAD59 D006
TARGET ADDRESS = 0x2002 0000
BYTE COUNT = 0x0000 0000
ARGUMENT = 0x0000 0000
Application 0 at 0x2000 0100
Application 1 at 0x2001 0000
Application 2 at 0x2002 0000

Figure 17-11. Multi-DXE Direct Code Execution Arrangement Example

Determining Boot Stream Start Addresses

The ROM functions `BFROM_MEMBOOT`, `BFROM_TWIBOOT`, `BFROM_SPIBOOT`, etc. not only allow the application to boot a subroutine residing at a given start address, they also assist in walking through linked multi-DXE streams.

When the `BFLAG_NEXTDXE` bit in `dFlags` is set and these functions are called, the system does not boot but instead walks though the boot stream following the next-DXE pointers. The `dBlockCount` parameter can be used to specify the DXE of interest. The routines then return the start address of the requested DXE's boot stream.

Initialization Hook Routine

When the ROM functions `BFROM_MEMBOOT`, `BFROM_SPIBOOT`, etc. are called, they create an instance of the `ADI_BOOT_DATA` structure on the stack and fill the items with default values. If the `BFLAG_HOOK` is set, the boot kernel invokes a callback routine which was passed as the fourth argument of the ROM routines, after the default values have been filled. The hook routine can be used to overwrite the default values. Every hook routine should fit the prototype:

```
void hook (ADI_BOOT_DATA* pBS);
```

The VisualDSP++ header files define the `ADI_BOOT_HOOK_FUNC` type the following way:

```
typedef void ADI_BOOT_HOOK_FUNC (ADI_BOOT_DATA*);
```

The hook function also gives access to the DMA load function used by the respective boot mode, which can be used for general purposes at runtime. For example, in the `BFROM_SPIBOOT` case, an instance of the load function:

```
ADI_BOOT_LOAD_FUNC *pSpiLoadFunction;
```

can be initialized by equipping the hook function with the instruction:

```
pSpiLoadFunction = pBS->pLoadFunction;
```

Specific Boot Modes

This section discusses individual boot modes and the required hardware connections.

The boot modes differ in terms of the booting source— for example whether data is loaded through the SPI or the parallel interface. Boot modes can also be grouped into slave boot modes and master boot modes.

In slave boot modes, the Blackfin processor functions as a slave to any host device, which is typically another embedded processor, an FPGA device or even a desktop computer. Likely, the Blackfin processor $\overline{\text{RESET}}$ input is controlled by the host device. So, usually the host sets $\overline{\text{RESET}}$ first, then waits until the preboot routine terminates by sensing the HWAIT output, and finally provides the boot data.

If a Blackfin processor, configured to operate in any of the slave boot modes, awakens from hibernate, it cannot boot by its own control. A feedback mechanism has to be implemented at the system level to inform the host device whether the processor is in hibernate state or not. The HWAIT strobe is an important primitive in such systems.

In the master boot modes, the Blackfin processor usually does not need to be synchronized and can load the boot data by itself. Master modes typically read from memory. This can be parallel memory such as flash devices, or serial memory that is read through SPI or TWI interfaces.

Memory boot modes should also be differentiated from peripheral boot modes. Boot modes that load boot streams through memory DMA are referred to as memory boot mode, reading data from regular memory. Peripheral modes load boot data through peripherals such as UART, TWI or SPI. With the exception of the FIFO boot, which is a hybrid, all memory boot modes are master modes. The boot source is typically non-volatile memory, such as a flash or EPROM device or even on-chip ROM. When supported by the system in warm boot scenarios, the boot source can also be SRAM or SDRAM.

Whether from the host (slave booting mode) or from memory (master booting mode), the boot source does not need to know about the structure of the boot stream. However in the case of Host DMA boot, the size (BYTE COUNT) of the boot stream should be known. This is because, having much more control over the Blackfin processor, the host must know what data is to be loaded to specific addresses.

No Boot Mode

When the BMODE pins are all tied low (BMODE = 0000), the Blackfin processor does not boot. Instead it processes factory-programmed OTP pages, then executes an IDLE instruction, preventing it from executing any instructions provided by the regular boot source. The purpose of this mode is to bring the processor up to a clean state after reset.

This mode helps to recover from malicious OTP configuration since it prevents execution of the user-controllable portion of the preboot routine.

When connecting an emulator and starting a debug session, the processor awakens from an idle due to the emulation interrupt and can be debugged in the normal manner.

The no boot mode is not the same as the bypass mode featured by the ADSP-BF53x Blackfin processor. To simulate that bypass mode feature using BMODE = 0001, see "Direct Code Execution" on page 17-34 and "Direct Code Execution" on page 17-148.

Flash Boot Modes

These booting modes are intended to boot from flash or EEPROM memories or even from battery-buffered SRAMs. The flash boot modes are

activated by BMODE = 0001. Although this is a single BMODE setting, the ADSP-BF52x Blackfin products support various configurations.

- Boot from 8-bit asynchronous flash memory
- Boot from 16-bit asynchronous flash memory

By default, the boot kernel does not alter any EBIU registers. Therefore, traditional asynchronous flash is assumed and maximum wait states are applied. By programming OTP half pages PBS00L and PBS00H, the user has the option to instruct the preboot routine to alter the EBIU registers as desired. In this way, the EBIU can be preset to access the flash device in either page mode or burst mode. There are also options to customize bus settings, such as wait states and ARDY behavior.

After the preboot routine returns and HWAIT is deasserted the first time, the boot kernel loads an initial burst of four 16-bit words. Then it interrogates the DMACODE field in the byte loaded from the 0x2000 0000 address. For flash mode, the DMA options shown in Table 17-9 are supported.

Table 17-9. DMA Options

DMACODE	DMA Width	Source Modify	Comment
1	8	1	Not recommended Provides ADSP-BF533 style 8-bit boot from 16-bit flash memory
2	8	2	8-bit MDMA boots from 8-bit flash mapped to lower byte of address bus.
6	16	2	16-bit MDMA boots from 16-bit flash
10	32	4	32-bit MDMA boots from 16-bit flash

The DMACODE field is filled by the elfloader utility based on boot mode, -width and -dmawidth settings. See the *VisualDSP++ Loader and Utility Manual* for details.

After the boot kernel has loaded and interpreted the first four 16-bit words, it continues loading the rest of the first block header and processes the boot stream.

Hardware configurations for the individual modes are shown in Figure 17-12 and Figure 17-13. The chip select is always controlled by the $\overline{\text{AMS0}}$ strobe. This maps the boot stream to the Blackfin processor's address 0x2000 0000.

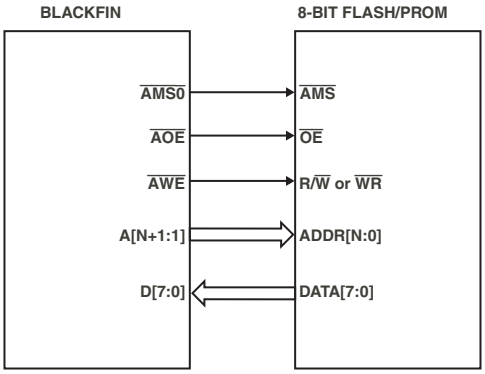


Figure 17-12. 8-Bit Flash Interconnection

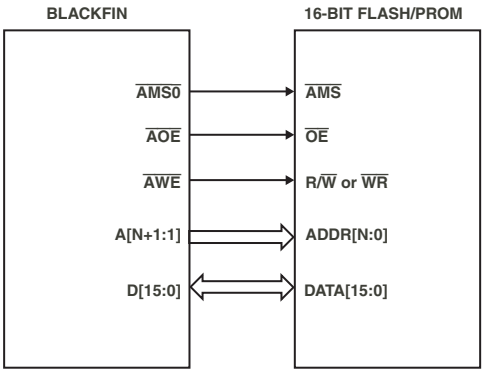


Figure 17-13. 16-Bit Flash Interconnection

Some flash devices provide write protection mechanisms, which can be activated during the power-up and reset cycles of the Blackfin processor. In the absence of such mechanisms, a pull-up resistor on the AMS0 strobe prevents the chip select from floating when the state of the processor is unknown.

The boot mode BMODE = 0001 can also be used to instruct the boot kernel to terminate immediately and directly execute code from the 16-bit flash memory instead. Code execution from 8-bit flash memory is not supported. See "Direct Code Execution" on page 17-34 for details.

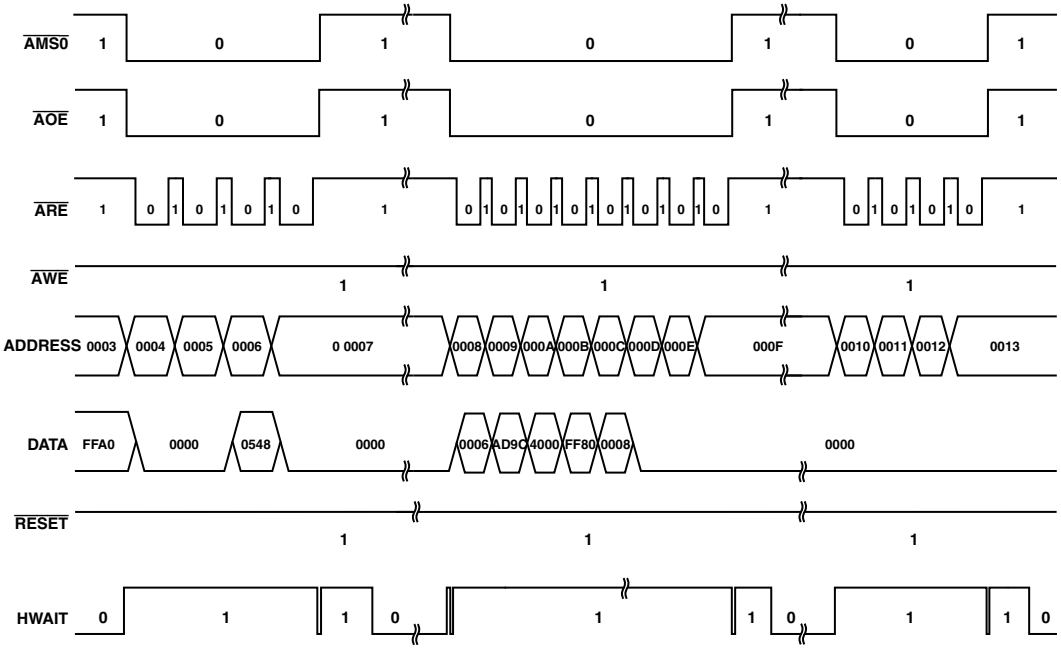


Figure 17-14. 16-bit Flash Mode Waveform

SDRAM Boot Mode

From the boot kernel perspective, the SDRAM boot mode is just another memory boot mode like flash boot. The only differences are that the boot stream is expected at address 0x0000 0010 and the initial eight bytes are loaded by two 32-bit loads.

From the application point of view, SDRAM boot is a completely different scheme. Since SDRAM is volatile memory, BMODE = 1010 is not a valid setting when the processor and the memories have just been powered up. This mode can only be used as a dynamically applied BMODE setting to install warm boot scenarios.

OTP programming is required to boot from SDRAM. Other boot modes can configure the SDRAM controller by execution of an initcode. But in the case of SDRAM boot, the initcode cannot be loaded without having the SDRAM controller already configured.

SDRAM boot is meaningful when the Blackfin processor is in hibernate state or is completely shut off for power savings while the SDRAM is kept alive in self-refresh mode.

Users who prefer to execute code out of SDRAM, rather than performing a boot from it, may refer to "Direct Code Execution" on page 17-34 for details.

FIFO Boot Mode

The FIFO boot mode (BMODE = 0010) boots the Blackfin processor from another processor or FPGA system, referred to as the host device. The host is decoupled from the Blackfin bus by an asynchronous FIFO memory. When compared to the glue-less Host DMA boot modes, the FIFO mode requires less intelligence from the host. The host device is only expected to handshake with the FIFO and to load the entire boot stream in 16-bit portions. There is no need for the host to know about the content and format of the boot stream.

The hardware configuration for the FIFO boot mode is shown in Figure 6-3 on page 6-40. The FIFO chip select connects to the $\overline{\text{AMS3}}$ strobe. Data read requests go to the $\overline{\text{DMAR1}}$ input on pin PG12. The host device controls the Blackfin processor's $\overline{\text{RESET}}$ input. As in all slave modes, the host device should not send requests to $\overline{\text{DMAR1}}$ unless the HWAIT signal goes inactive. The host device may optionally rely on HWAIT edges to continue or discontinue transmission of boot data in an interrupt controlled manner.

From the boot kernel perspective the FIFO boot mode (BMODE = 0010) is just another memory boot mode, the only exception being that the HMDMA1 block is enabled in advance. Activating this functionality makes the FIFO boot mode become a slave mode.

The bits set in the HMDMA1_CONTROL register are REP and HMDMAEN. Since the ADSP-BF52x products do not support the SND bit as the ADSP-BF54x products do, the FIFO boot mode requires further precautions.

In the FIFO boot mode, the DMACODE field in the boot block headers must always be 0x06, which instructs the boot kernel to perform 16-bit DMA. The boot kernel increments the applied addresses as if reading from flash memory.

Regardless of the HMDMA settings, the source channel of the memory DMA prefetches four 32-bit words as soon as enabled. Only the transmit channel is stalled and triggered by the HMDMA module. In 16-bit DMA mode, these four early reads translate to eight 16-bit reads.

The ADSP-BF52x boot kernel ensures that at least 16 valid data words are ready in the external FIFO—by first counting eight rising edges on the $\overline{\text{DMAR1}}$ request input and then disabling the HMDMA module. When HMDMA is later re-enabled, the prefetch will find valid data and the MDMA can be started safely.

This method requires that the host send 16 more request strobes after it has sent the complete boot stream to the FIFO. This is because the trans-

mit channel of the DMA still has to drain the FIFO, which must be protected from underflow at start.

SPI Master Boot Modes

The SPI boot mode (BMODE = 0011) boots from SPI memories connected to the SPISEL1 interface. 8-, 16-, 24-, and 32-bit address words are supported. Standard SPI memories are read using either the standard 0x03 SPI read command or the 0x0B SPI fast read command.

Unlike other Blackfin processors, the ADSP-BF52x Blackfin processors have no special support for DataFlash devices from Atmel. Nevertheless, DataFlash devices can be used for booting and are sold as standard 24-bit addressable SPI memories. They also support the fast read mode. If used for booting, DataFlash memory must be programmed in the power-of-2 page mode.

For booting, the SPI memory is connected as shown in Figure 17-15.

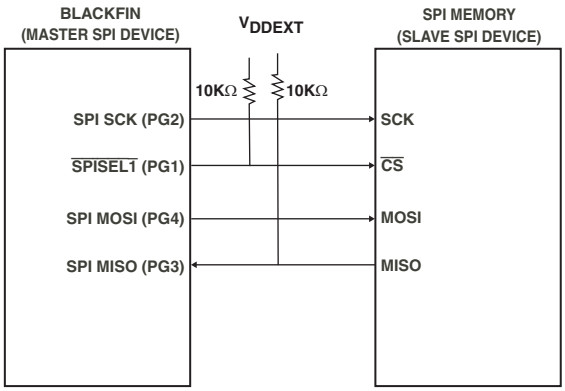


Figure 17-15. Blackfin to SPI Memory Connections

The pull-up resistor on the MISO line is required for automatic device detection. The pull-up resistor on the SPISEL1 line ensures that the memory is in a known state when the Blackfin GPIO is in a high-impedance

state (for example, during reset). A pull-down resistor on the SPISCK line displays cleaner oscilloscope plots during debugging.

For SPI master boot, the SPE, MSTR and SZ bits are set in the SPI_CTL register. For details see Chapter 22, "SPI-Compatible Port Controller". With TIMOD = 2, the receive DMA mode is selected. Clearing both the CPOL and CPHA bits results in SPI mode 0. The boot kernel does not allow SPI hardware to control the $\overline{\text{SPISEL1}}$ pin. Instead, this pin is toggled in GPIO mode by software. Initialization code is allowed to manipulate the uwSsel variable in the ADI_BOOT_DATA structure to extend the boot mechanism to a second SPI memory connected to another GPIO pin.

By default, the boot kernel sets the SPI_BAUD register to a value of 133, resulting in a bit rate of SCLK/266. This default value can be altered by programming the 4-bit OTP_SPI_BAUD field in OTP page PBS00L to one of the values in Table 17-10.

Table 17-10. Bit Rate

OTP_SPI_BAUD	SPI_BAUD	Bit Rate
b#0000	133	SCLK/(2x133)
b#0001	Reserved	
b#0010	2	SCLK/(2x2)
b#0011	4	SCLK/(2x4)
b#0100	8	SCLK/(2x8)
b#0101	16	SCLK/(2x16)
b#0110	32	SCLK/(2x32)
b#0111	64	SCLK/(2x64)

Similarly, the boot kernel uses the standard 0x03 SPI read command, by default. Programming the OTP_SPI_FASTREAD bit in OTP page PBS00L enables the fast read mode where the boot kernel uses the 0x0B read command instead and transmits a dummy zero byte after the address bytes.

SPI Device Detection Routine

Since BMODE = 0011 supports booting from various SPI memories, the boot kernel automatically detects what type of memory is connected. To determine whether the SPI memory device requires an 8-, 16-, 24- or 32-bit addressing scheme, the boot kernel performs a device detection sequence prior to booting. The MISO signal requires a pull-up resistor, since the routine relies on the fact that memories do not drive their data outputs unless the right number of address bytes are received.

Initially, the boot kernel transmits a read command (either 0x03 or 0x0B) on the MOSI line, which is immediately followed by two zero bytes. Once the transmission is finished, the boot kernel interrogates the data received on the MISO line. If it does not equal 0xFF (usually a DMACODE value of 0x01 is expected), then an 8-bit addressable device is assumed.

If the received value equals 0xFF, it is assumed that the memory device has not driven its data output yet and that the 0xFF value is due to the pull-up resistor. Thus, another zero byte is transmitted and the received data is tested again. If it differs from 0xFF, either a 16-bit addressable device (standard mode) or an 8-bit addressable device (fast read mode) is assumed.

If the value still equals 0xFF, device detection continues. Device detection aborts immediately if a byte different than 0xFF is received. The boot kernel continues with normal boot operation and it re-issues a read command to read from address 0 again. The first block header is loaded by two read sequences, further block headers and block payload fields are loaded by separate read sequences.

Figure 17-16 illustrates how individual devices would behave.

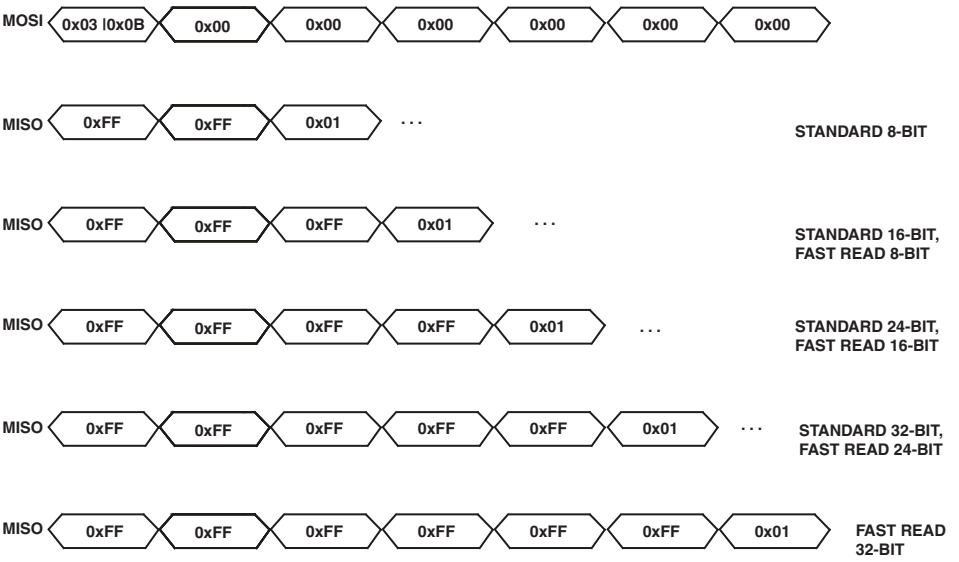


Figure 17-16. SPI Device Detection Principle

Figure 17-17 on page 17-69 shows the initial signaling when a 24-bit addressable SPI memory is connected in SPI master boot mode. After $\overline{\text{RESET}}$ releases and preboot has processed relevant OTP pages, a 0x03 command is transmitted to the MOSI output, followed by a number of 0x00 bytes. The 24-bit addressable memory device returns a first data byte at

the fourth zero byte. Then, the device detection has completed and the boot kernel re-issues a 0x00 address to load the boot stream.

Figure 17-17. Typical SPI Master Boot Waveforms

SPI Slave Boot Mode

For SPI slave mode boot (BMODE = 0100), the Blackfin processor is consuming boot data from an external SPI host device. The hardware configuration is shown in Figure 17-18. As in all slave boot modes, the host device controls the Blackfin processor RESET input.

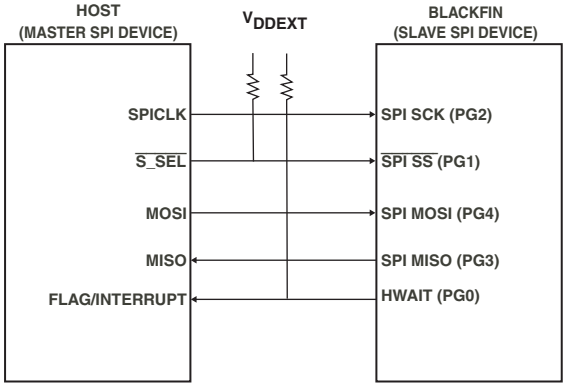


Figure 17-18. Connections Between Host (SPI Master) and Blackfin Processor (SPI Slave)

The host drives the SPI clock and is responsible for the timing. The host must provide an active-low chip select signal that connects to the SPISS

input of the Blackfin processor. It can toggle with each byte transferred or remain low during the entire procedure. 8-bit data is expected. The 16-bit mode is not supported.

In SPI slave boot mode, the boot kernel sets the CPHA bit and clears the CPOL bit in the SPI_CTL register. Therefore the MISO pin is latched on the falling edge of the MOSI pin. For details see Chapter 22, "SPI-Compatible Port Controller".

In SPI slave boot mode, HWAIT functionality is critical. When high, the resistor shown in Figure 17-18 programs HWAIT to hold off the host. HWAIT holds the host off while the Blackfin processor is in reset or executing the preboot. Once HWAIT turns inactive, the host can send boot data. The SPI module does not provide very large receive FIFOs, so the host must test the HWAIT signal for every byte. Figure 17-20 on page 17-71 illustrates the required program flow on the host side.

Figure 17-19 on page 17-70 shows the initial waveform for an SPI slave boot case. As soon as the Blackfin processor releases HWAIT after reset, the host device pulls the $\overline{\text{SPISS}}$ pin low and starts transmitting data. After the eighth data word has been received, the boot kernel asserts HWAIT again as it has to process the DMACODE field of the first block header. When the host detects the asserted HWAIT it gracefully finishes the transmission of the on-going word. Then, it pauses transmission until HWAIT releases again.

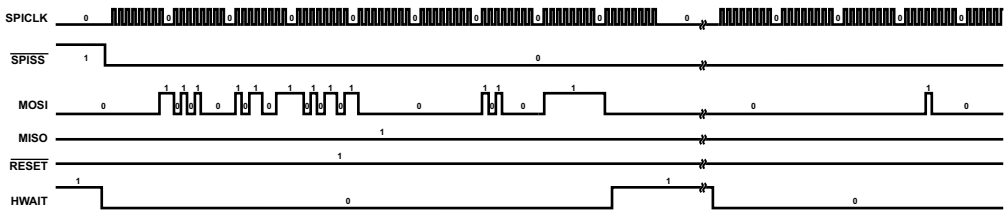


Figure 17-19. Typical SPI Slave Boot Waveforms

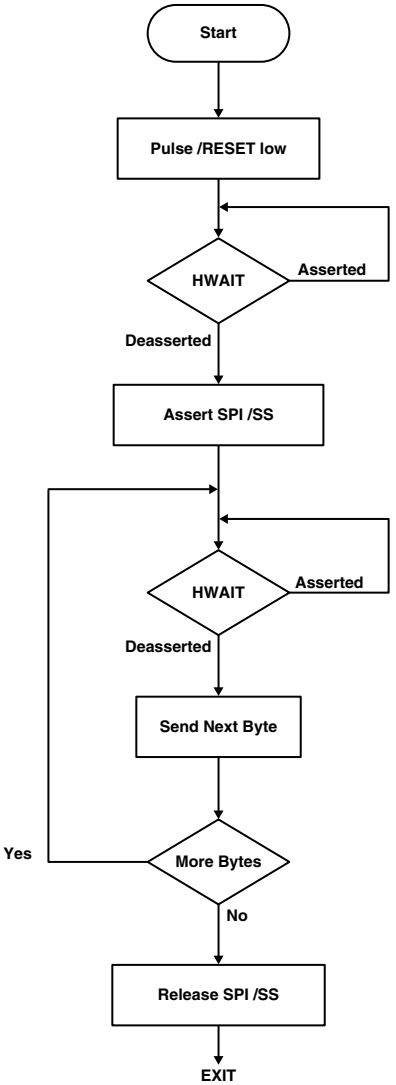


Figure 17-20. SPI Program Flow on Host Device

TWI Master Boot Mode

In TWI master boot mode (BMODE = 0101) the boot kernel reads boot data from I^2C memory connected to the TWI interface. The Blackfin processor selects the slave EEPROM with the unique ID 0xA0, submits successive read commands to the device starting at internal address 0x0000, and begins clocking data to the processor. The EEPROM's device select bits A2–A0 must be 0s (tied low) when present. The I^2C EPROM device should comply with Philips I^2C Bus Specification version 2.1 and should have the capability to auto increment its internal address counter such that the contents of the memory device can be read sequentially. Connections are shown in Figure 17-21 "TWI Master Boot Mode Connections" on page 17-73.

On the Blackfin processor, in both TWI master and slave boot modes, the upper 512 bytes starting at address 0xFF90 3E00 either must not be used or must be booted last. The boot ROM code uses this space for the TWI boot modes to temporarily hold the serial data which is then transferred to L1 instruction memory using DMA. All boot blocks that target the L1 instruction memory or external memories must have the BFLAG_INDIRECT bit set. Initcodes

can alter the placement of the temporary buffer by modifying the `pTempBuffer` and `dTempByteCount` variables in the `ADI_BOOT_DATA` structure.

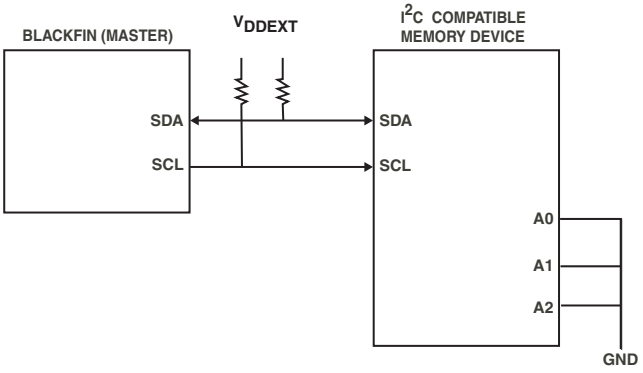


Figure 17-21. TWI Master Boot Mode Connections

The Blackfin processor's TWI controller outputs the address of the I^2C device to boot from, in this case 0xA0, where the least significant bit indicates the direction of the transfer. In this example, it is a write (0) to write the first two bytes of the internal address from which to start booting (0x00).

Figure 17-23 "TWI Init and Fill Block Timing" on page 17-74 shows the TWI init and zero fill blocks.

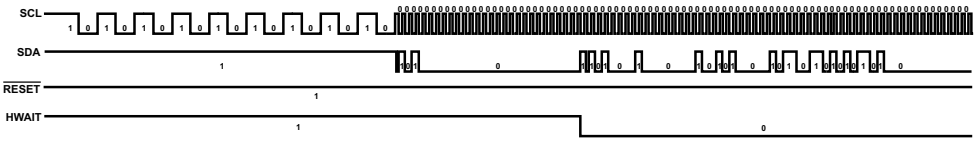


Figure 17-22. TWI Master Boot Timing

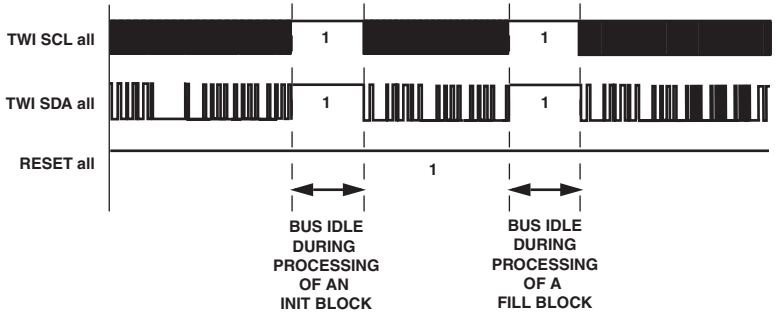


Figure 17-23. TWI Init and Fill Block Timing

Figure 17-22 “TWI Master Boot Timing” on page 17-73 shows the initial waveforms for TWI master boot. After reset, the kernel generates nine slow pulses on the SCL output to ensure the TWI memory's state machine exits any pending state. Then a start condition is issued and 0xA0 address command is issued, where the least significant bit indicates the direction of the write. In this case it is a write (0) in order to write two more 0x00 address bytes.

By default, it is assumed that the I^2C memory device is two-byte addressable. This can be changed by programming the `OTP_TWI_TYPE` bit field in OTP page `PBS00L` as shown in Table 17-11 and Figure 17-44 on page 17-107.

Table 17-11. Addressable Bytes

OTP_TWI_TYPE	Address Bytes
00	2
01	3
10	4
11	1

The TWI controller is programmed to generate a 30% duty cycle clock in accordance with the I^{2}C clock specification for fast-mode operation (PRESCALE = 0xA, CLKDIV = 0x811) as shown in Table 17-12. The default values can be altered by OTP programming. Setting the OTP_TWI_CLKDIV bit in OTP page PBS00L changes the OTP_TWI_CLKDIV register value to 0x3232 as recommended for 100 kHz TWI operation. The OTP_TWI_PRESCALE field controls the prescale value written to the TWI_CONTROL register.

Table 17-12. Prescale Value

OTP_TWI_PRESCALE	PRESCALE	Recommended[1]
000	0x0A	SCLK = 100 MHz
001	0x0E	SCLK = 140 MHz (theoretical)
010	0x0C	SCLK = 120 MHz
011	0x0A	SCLK = 100 MHz
100	0x08	SCLK = 80 MHz
101	0x06	SCLK = 60 MHz
110	0x04	SCLK = 40 MHz
111	0x02	SCLK = 20 MHz

1 Check the appropriate data sheet for the maximum SCLK frequency.

TWI Slave Boot Mode

In TWI slave boot mode (BMODE = 0110) the Blackfin processor consumes data from a I^{2}C host device connected to the TWI interface. The I^{2}C host selects the slave (Blackfin processor) with the 7-bit slave address 0x5F. When the Blackfin processor acknowledges, the host can download the boot stream. The I^{2}C host should comply with Philips I^{2}C Bus Specification version 2.1. The host supplies the serial clock.

Connections are shown in Figure 17-24 “TWI Slave Boot Mode Connections” on page 17-76.

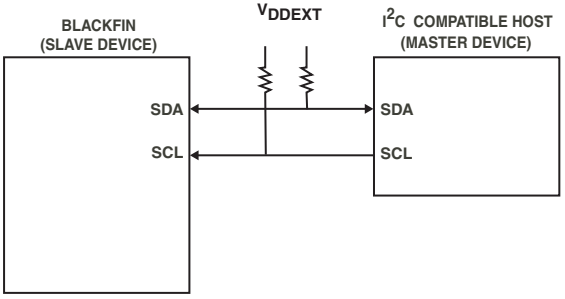


Figure 17-24. TWI Slave Boot Mode Connections

Figure 17-25 “TWI Slave Boot Timing” on page 17-76 shows initial waveforms for TWI slave boot. As soon as HWAIT releases after reset the host starts transmitting the boot stream data. It starts with a start condition and a 0xBE command, which is a composite of the 0x5F address and a trailing zero bit to indicate write direction.

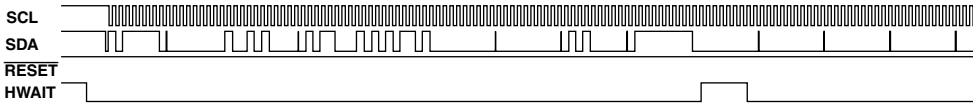


Figure 17-25. TWI Slave Boot Timing

Figure 17-26 on page 17-77 shows an example of bit stretching.

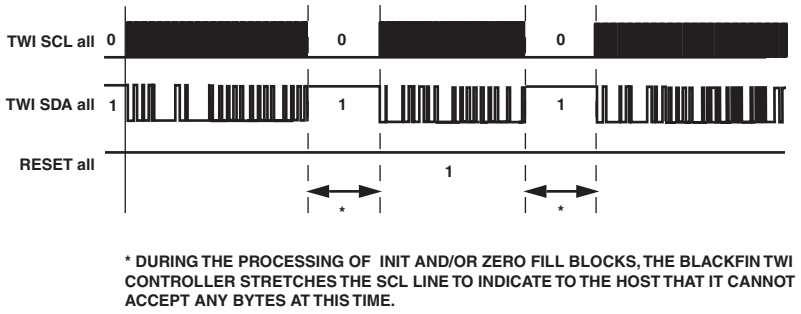


Figure 17-26. TWI Bit Stretching Timing

On the Blackfin processor, in both TWI master and slave boot modes, the upper 512 bytes starting at address 0xFF90 3E00 either must not be used or must be booted last. The boot ROM code uses this space for the TWI boot modes to temporarily hold the serial data which is then transferred to L1 instruction memory using DMA. All boot blocks that target the L1 instruction memory or external memories must have the `BFLAG_INDIRECT` bit set. Initcodes can alter the placement of the temporary buffer by modifying the `pTempBuffer` and `dTempByteCount` variables in the `ADI_BOOT_DATA` structure.

UART Slave Mode Boot

Figure 17-27 on page 17-78 shows the interconnection required for booting. The figure does not show physical line drivers and level shifters that are typically required to meet the individual UART-compatible standards.

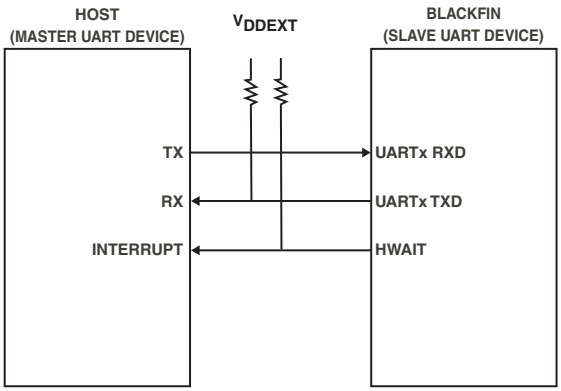


Figure 17-27. UART Slave Boot Mode Connections

For BMODE = 0111, the ADSP-BF522/523/524/525/526/527 processor consumes boot data from a UART host device connected to the UART0 interface on port G. For BMODE = 1000 the ADSP-BF522/523/524/525/526/527 processor consumes boot data from a UART host device connected to the UART1 interface on port F. Hardware flow control is simulated by the HWAIT signal.

The host downloads programs formatted as boot streams using an autobaud detection sequence. The host selects a bit rate within the UART clocking capabilities. To determine the bit rate when performing the autobaud, the boot kernel expects an "@" character (0x40, eight data bits, one start bit, one stop bit, no parity bit) on the UART RXD input. The boot kernel acknowledges, and the host then downloads the boot stream. The acknowledgement consists of four bytes: 0xBF, UARTx_DLL, UARTx_DLH, 0x00. The host is requested to not send further bytes until it has received the complete acknowledge string. Once the 0x00 byte is received, the host

can send the entire boot stream. The host should know the total byte count of the boot stream, but it is not required to have any knowledge about the content of the boot stream. Further information regarding auto-baud detection is given in “Autobaud Mode” on page 10-31.

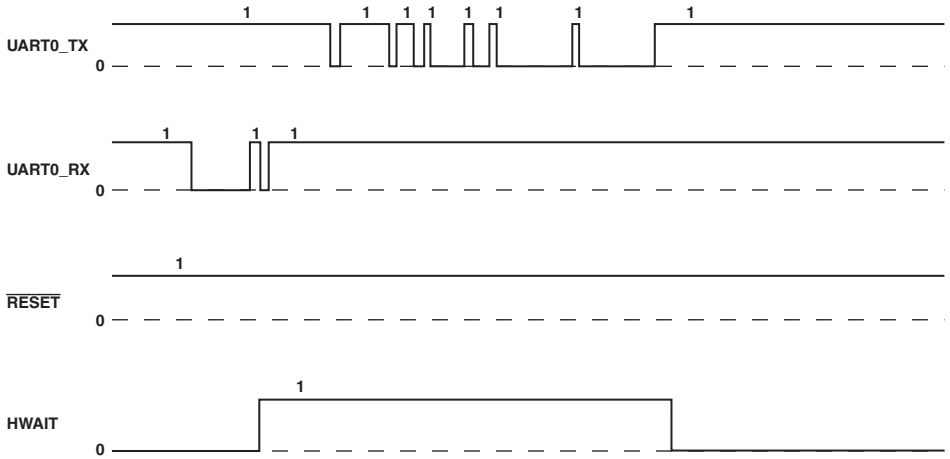


Figure 17-28. UART Autobaud Waveform

When the boot kernel is processing fill or initcode blocks it might require extra processing time and needs to hold the host off from sending more data. This is signalled with the HWAIT output. When equipped with a pull-up resistor the HWAIT signal imitates the behavior of an RTS output and could be connected to the CTS input of the booting host. The host is not allowed to send data until HWAIT turns inactive after a reset cycle. Therefore a pulling resistor on the HWAIT signal is required.

If the resistor pulls to ground, the host must pause transmission when HWAIT is low and is permitted to send when HWAIT is high. A pull-up resis-

tor inverts the signal polarity of HWAIT. The host should test HWAIT at every transmitted byte.

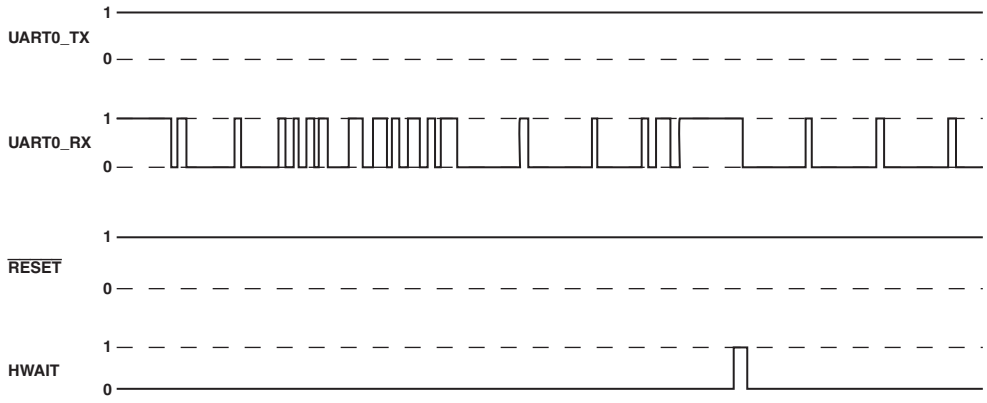


Figure 17-29. UART Boot

Figure 17-29 on page 17-80 shows the initial case of the UART boot mode. As soon as HWAIT releases after reset, the boot kernel expects to receive a 0x40 byte for bit rate detection. After the bit rate is known, the UART is enabled and the kernel transmits for bytes.

For UART boot, it is not obvious on how to change the PLL by an initcode routine. This is because the UARTx_DLL and UARTx_DLH registers have to be updated to keep the required bit rate constant after the SCLK frequency has changed. It must be ensured that the host does not send data while the PLL is changing. The initcode examples provided along with the VisualDSP++ tools installation demonstrate how this can be accomplished.

OTP Boot Mode

In the OTP boot mode (BMODE = 1011), the boot kernel loads the boot stream from the on-chip OTP memory. OTP booting is a self-sufficient

booting mechanism that does not require external boot memory or a host device.

By default the boot kernel starts loading the boot stream starting from OTP page 0x40. This is in the public OTP region. The boot stream can occupy all pages up to OTP page 0xDF, resulting in a boot stream length of up to 2560 bytes. The start address of the boot stream can be altered by programming the OTP_START_PAGE field in the PBS01H page. If there is no conflict with the alternate preboot pages feature, the OTP_START_PAGE field can be set to 0x20, resulting in a boot stream length of up to 3072 bytes.

In the current implementation, the OTP engine has no DMA support. Data is loaded and copied by core instructions. Nevertheless the DMACODE field should be set to 0xA, indicating 32-bit operation. The boot kernel ensures proper operation at 32-bit granularity, but 64-bit alignment may help to reduce the number of OTP pages that have to be read during boot processing. Byte 0 of the boot stream is expected to be byte 0 of the lower 32-bit word of the lower 0x40 half page.

In the OTP boot mode, the upper 512 bytes starting at address 0xFF90 3E00 either must not be used or must be booted last. The boot ROM code uses this space to temporarily hold the serial data which is then transferred to L1 instruction memory using DMA. All boot blocks that target the L1 instruction memory or external memories must have the BFLAG_INDIRECT bit set. Initcodes can alter the placement of the temporary buffer by modifying the pTempBuffer and dTempByteCount variables in the ADI_BOOT_DATA structure.

Host DMA Boot Modes

The Host DMA boot modes differ completely from other boot modes because the boot kernel has no control over the DMA channels. The host device masters the DMA, so the host device must parse the boot stream by itself.

The two host DMA boot modes (BMODE = 1110 for 16-bit and BMODE = 1111 for 8-bit) are almost identical. The differences are the port muxing control and the initial programming of the HOST_CONTROL register. The 16-bit boot mode uses the HOSTDP's acknowledge mode while the 8-bit boot mode sets the INT_MODE bit in the HOST_CONTROL register to activate the interrupt mode.

Connection of a host device to the Blackfin processor is discussed in Chapter 8, "Host DMA Port". For booting, the host device should control the $\overline{\text{RESET}}$ of the Blackfin processor. The host processor must poll the HOST_STATUS register using a configuration read of the HOSTDP until the ALLOW_CNFG bit is set (indicating that the host may begin sending the 7 configuration words). This is necessary before each configuration of the HOSTDP. The host processor may optionally sense the HWAIT signal to determine when it should begin polling the ALLOW_CNFG bit.

The HOSTDP interface does not support the advanced boot kernel operations such as fill, CRC or callback. There is simple support to simulate the initcode functionality. Typically, this feature is not so important when the preboot OTP memory pages can be programmed to configure the PLL and SDRAM controllers. However, if the user does not have the option to program OTP memory, the simulated initcode is the only option to speed up the processor clocks and to enable the SDRAM controller for booting. One of these options must be used for the host device to boot into SDRAM memory.

In order to simulate initcodes the host device must send a valid initcode routine to L1 instruction address 0xFFA0 0000. Additionally, the host is required to issue an HIRQ command after sending the 7 configuration words (but before sending any data) for the initcode block to the HOSTDP. Once the boot kernel detects an HIRQ command from the host and the DMA work unit is complete, the boot kernel will issue a CALL instruction to the address held in the EVT1 register, and the C language initcode routine is called. EVT1 defaults to 0xFFA0 0000, but it can be modified by user instructions during the boot process. When the initcode

returns, the regular boot process continues. This can be repeated multiple times if necessary.

If the initcode routine has properly configured the SDRAM controller, subsequent Host DMA work units can write to SDRAM memory. Similarly, if the initcode has programmed the PLL, the Host DMA port can run at higher speed since it is `SCLK` dependent.

The same scheme is used to terminate the boot process. When the host is ready to send the final boot block of the application it needs to send the 7 configuration words required by the HOSTDP. The host device should then send an `HIRQ` command followed by the remaining data. Once all data has written, the boot kernel executes another `CALL` instruction and the application takes control of the system rather than returning to the boot kernel.

Figure 17-30 on page 17-84 illustrates boot kernel processing in the Host DMA boot mode. Figure 17-31 on page 17-85 illustrates host device flow.

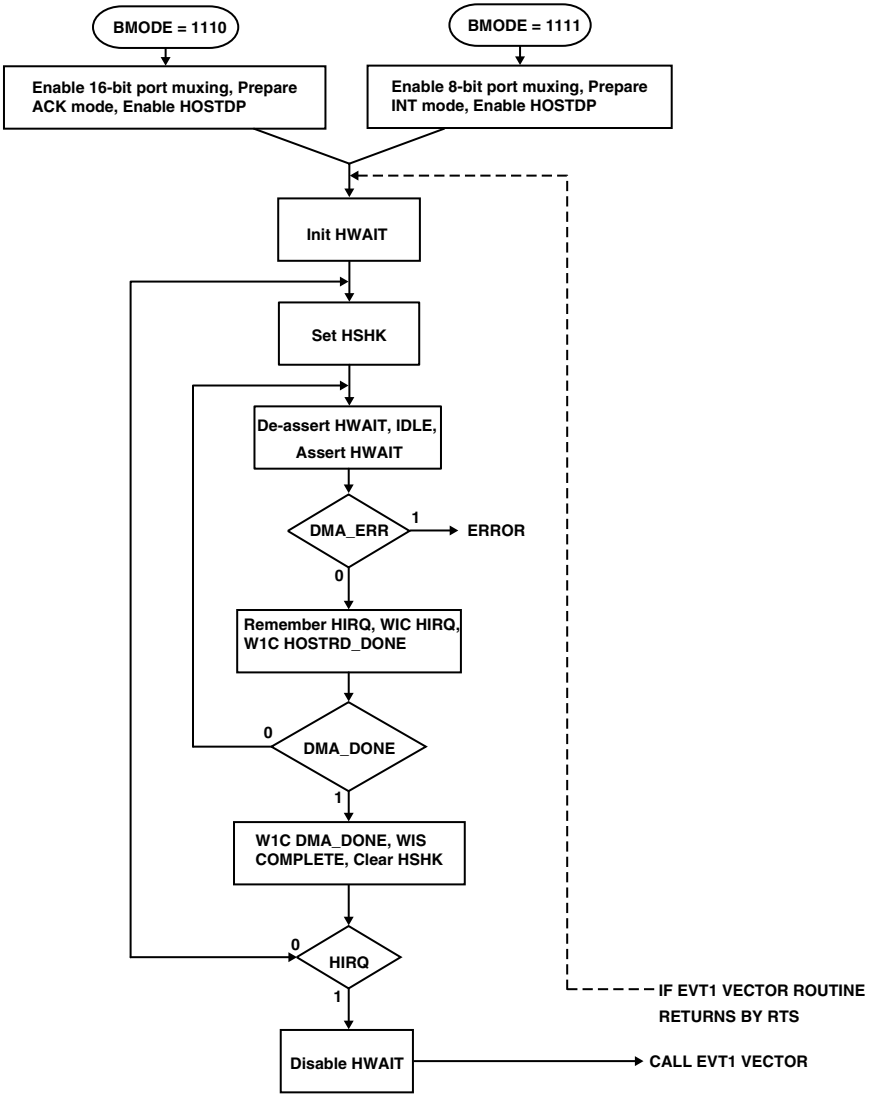


Figure 17-30. Boot Kernel Processing in Host DMA Boot Mode

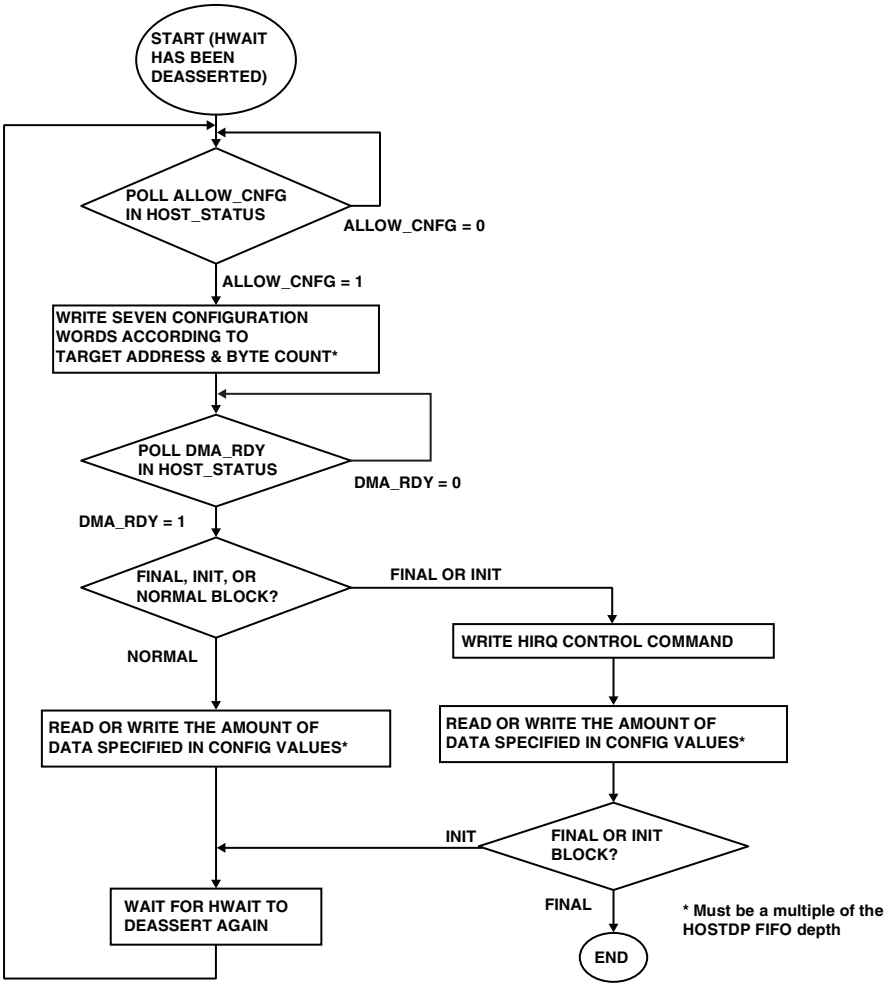

START (HWAIT HAS BEEN DEASSERTED)
POLL ALLOW_CNFG IN HOST_STATUS
ALLOW_CNFG = 0
ALLOW_CNFG = 1
WRITE SEVEN CONFIGURATION WORDS ACCORDING TO TARGET ADDRESS & BYTE COUNT*
POLL DMA_RDY IN HOST_STATUS
DMA_RDY = 0
DMA_RDY = 1
FINAL, INIT, OR NORMAL BLOCK?
FINAL OR INIT
NORMAL
WRITE HIRQ CONTROL COMMAND
READ OR WRITE THE AMOUNT OF DATA SPECIFIED IN CONFIG VALUES*
READ OR WRITE THE AMOUNT OF DATA SPECIFIED IN CONFIG VALUES*
INIT
FINAL OR INIT BLOCK?
FINAL
WAIT FOR HWAIT TO DEASSERT AGAIN
END
* Must be a multiple of the HOSTDP FIFO depth

Figure 17-31. Host Device Flow in Host DMA Boot Mode

NAND Flash Boot Mode

NAND flash boot mode (BMODE = 1100 and 1101) is intended to boot from SLC NAND flash memory devices connected directly to the NAND Flash Controller (NFC) of the ADSP-BF52x processors.

By default the NAND flash boot mode configures the read and write delay strobe timing parameters within the NFC_CTL register with RD_DLY = 0x3 and WR_DLY = 0x3. This provides t_{RP} and t_{WP} timings of four SCLK cycles (30ns at 133 MHz) to provide maximum compatibility. By programming OTP half page PBS01H, the user has the option to instruct the preboot routine to provide alternate settings prior to accessing the NAND flash for the first access. In NAND flash boot mode, the HWAIT signal does not toggle. The respective GPIO pins remain in high-impedance mode.

Providing OTP configurations of RD_DLY = 0x0 and WR_DLY = 0x0 will result in the boot kernel using the default configuration of RD_DLY = 0x3 and WR_DLY = 0x3. The highest performance settings for NAND flash boot are enabled with WR_DLY = 0x1 and RD_DLY = 0x0.

Supported Devices

NAND flash boot provides support for booting from a number of NAND flash devices from a number of different manufacturers. There are two main classifications of single SLC NAND flash memories:

- small-page NAND flash
- large-page NAND flash (8-bit)

The small-page NAND flash devices use a different addressing scheme for accessing the NAND flash array than that required by large-page NAND flash devices. Additionally small-page devices require a different command set for reading from different parts of the page.

For booting, small-page devices must comply with the array configuration in Table 17-13 and support the commands in Table 17-14.

Table 17-13. Supported Small-page Device Array Configuration

Parameter	Size
Page Size	512 Bytes
Block Size (excluding spare area)	16384 Bytes (32 pages)
Spare Area	16 Bytes
1st half of page	256 Bytes
2nd half of page	256 Bytes
Maximum number of addressable blocks	524288

Table 17-14. Supported Small-page Commands

Operation	Command
Reset	0xFF
Read from 1st half of array	0x00
Read from 2nd half of array	0x01
Read from spare area	0x50

The NAND flash boot kernel, by default, issues four address cycles after issuing the read command. This redundancy can be removed by modifying the `uwNumCommands` parameter of the `ADI_BOOT_NAND_ADDRESS` structure in an initialization routine that is executed before the main application boot stream is processed. To load the initialization function however, four address cycles are always issued. The NAND flash device must be capable of ignoring the additional address cycles.

NAND flash boot provides support for a number of large-page array configurations. The fourth byte of the NAND flash Electronic Signature is used to configure the boot kernel for correct access to the memory array.

The boot kernel supports any large-page NAND flash device whose Electronic Signature fourth byte is complies with the format in Figure 17-32.

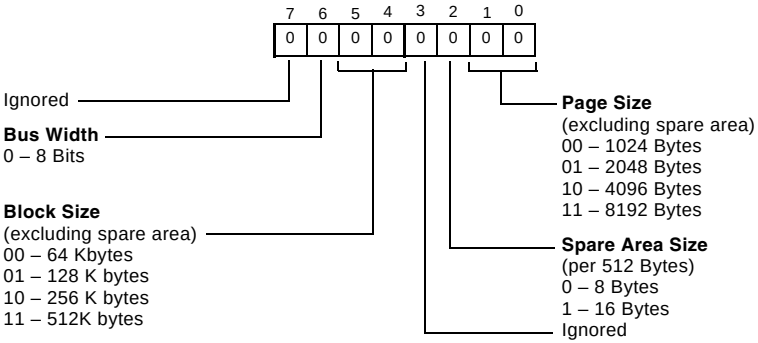


Figure 17-32. Supported 4th byte of NAND Flash Electronic Signature

Table 17-15 shows the large-page command set that is supported.

Table 17-15. Supported Large-page Commands

Operation	1st Command	2nd Command
Reset	0xFF	
Read Electronic Signature	0x90	
Read	0x00	0x30

Due to the auto detection method used, large-page NAND flash devices must not react to the issuing of command 0x50 followed by four address cycles by driving the `R B` signal low and then high again.

The NAND flash boot kernel, by default, issues five address cycles after issuing the read command. This redundancy can be removed by modifying the `uwNumCommands` parameter of the `ADI_BOOT_NAND_ADDRESS` structure in an initialization routine that

is executed before the main application boot stream is processed. To load the initialization function however, five address cycles are always issued. The NAND flash device must be capable of ignoring the additional address cycles.

Hardware configuration for the NAND flash boot mode is shown in Figure 17-33.

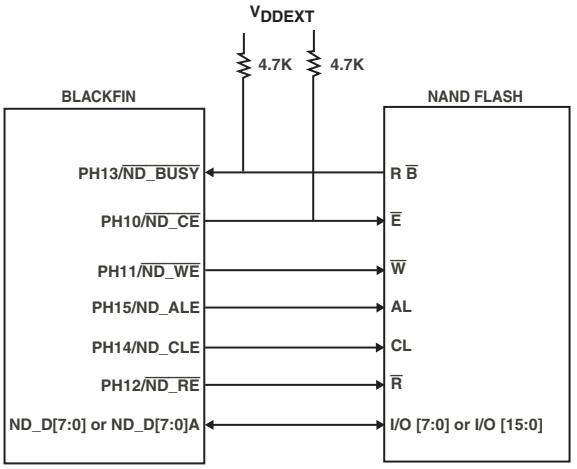


Figure 17-33. 8-Bit NAND Flash Interconnection

As the GPIO pins were originally configured as inputs, a pull-up resistor is required on PH10\ND_CE. This ensures that the device is not selected after a reset before the required GPIO pins have been configured correctly.

NAND Flash Page Structure

The NAND flash boot option transfers data contents from the main area of the NAND flash using a 256 byte DMA transfer. The spare area section at the end of the page contains the ECC error checking parity data for each 256 sub block of the main data area. The spare area section is divided into equal sizes corresponding to the number of 256 byte blocks contained

within a page. For a 512 Mbyte small-page device, the spare area is divided into two sections.

The first three bytes of each sub section of the spare area contains the 22-bit ECC parity data for the corresponding data block. The very last byte of the spare area is reserved in the first and second pages of each block for the bad-block marker. Figure 17-34 shows the page structure for a NAND flash device with a page size of 2048-bytes. The 64-byte area is divided into eight 8-byte sections. The first three bytes of each section contains the parity data for the corresponding 256-byte block.

The last byte in the page is used as the bad-block marker in the event that the device only contains 8-bytes of spare area per 512-byte block instead of the more common 16-bytes per 512-byte block.

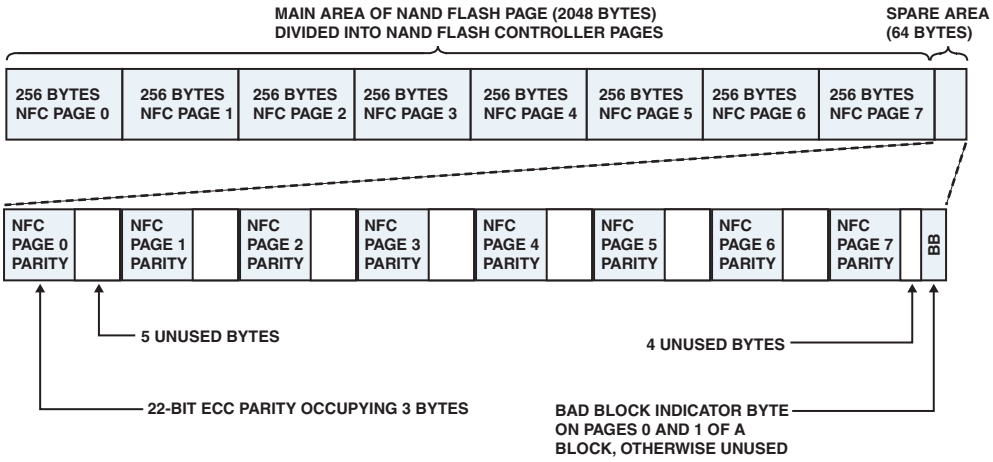

MAIN AREA OF NAND FLASH PAGE (2048 BYTES)
DIVIDED INTO NAND FLASH CONTROLLER PAGES
SPARE AREA
(64 BYTES)
256 BYTES NFC PAGE 0
256 BYTES NFC PAGE 1
256 BYTES NFC PAGE 2
256 BYTES NFC PAGE 3
256 BYTES NFC PAGE 4
256 BYTES NFC PAGE 5
256 BYTES NFC PAGE 6
256 BYTES NFC PAGE 7
NFC PAGE 0 PARITY
NFC PAGE 1 PARITY
NFC PAGE 2 PARITY
NFC PAGE 3 PARITY
NFC PAGE 4 PARITY
NFC PAGE 5 PARITY
NFC PAGE 6 PARITY
NFC PAGE 7 PARITY
BB
5 UNUSED BYTES
4 UNUSED BYTES
22-BIT ECC PARITY OCCUPYING 3 BYTES
BAD BLOCK INDICATOR BYTE ON PAGES 0 AND 1 OF A BLOCK, OTHERWISE UNUSED

Figure 17-34. Page Layout of NAND Flash Device, 2048-Byte Page Size

Auto Detection

Once the boot kernel has detected the NAND flash boot option, the first operation to be performed is the auto detection procedure of the NAND flash device.

The boot kernel first issues a reset signal to the NAND flash device. The reset command brings the NAND flash out of the default read mode—ready to accept a command. The NAND flash reacts by driving the R $\overline{B}$ signal low and then high again.

The processor, after issuing the reset command, enters a nested loop that checks the status of the R $\overline{B}$ signal every 100 SCLK cycles. A maximum of 100 checks are performed. If the ready busy signal is not driven low and then high again after 100 attempts, then the boot kernel enters the safe idle mode as it assumed that no NAND flash device is present. The loop terminates when R $\overline{B}$ assertion is detected. The processor then proceeds to determine if the attached device is a small-page NAND flash device.

Small-page device detection consists of issuing a command to read from the spare area of the device, command 0x50, followed by four address cycles. Once again the processor enters a nested loop routine waiting for detection of a rising edge of the $\overline{\text{ND_BUSY}}$signal. If the rising edge is detected then the boot kernel is configured to boot from the supported small-page device.

If no rising edge is detected by the time the loop terminates, the device is assumed to be a large-page device. The processor issues another reset command to reset the large-page device, then proceeds to read the Electronic Signature to configure the boot kernel appropriately.

Boot Stream Processing

To successfully boot from NAND flash, blocks of data must be first transferred to the processors internal memory to be processed by the boot

kernel. A 512 byte temporary storage space located at 0xFF907E00 – 0xFF907FFF is used.

This storage space is split into two buffers each consisting of 256 bytes.

The 256 byte buffer at location 0xFF907E00 – 0xFF907EFF is referred to as the “MainBuffer”. The remaining 256 bytes from 0xFF907F00 – 0xFF907FFF are referred to as the “PrefetchBuffer”.

The NAND flash controller is configured for a 256 byte page size. During the boot phase, a single block transfer consists of 256 bytes. All block transfers from the NAND flash device go the PrefetchBuffer. The boot kernel determines if the MainBuffer is empty, is partially processed or is fully processed. If the MainBuffer is empty or all data currently residing in the MainBuffer is processed, the boot kernel copies the contents of the PrefetchBuffer into the MainBuffer—then requests another 256 block of data from the NAND flash. This process continues until the entire boot stream is processed.

An important requirement of NAND flash devices is the need for error checking and correction (ECC) on the received data. The NAND flash controller of the ADSP-BF52x devices uses a Hamming Code algorithm to automatically generate two sets of parity data for each 256 byte block transfer. The two sets of parity data each consist of 11 bits providing a total of 22 bits of parity data. This allows for the detection and correction of a single bit error within a 256 byte block, detection of a double error, and detection of an error within the parity data itself.

The boot kernel uses the embedded NFC ECC parity generation hardware and performs the error correction algorithm after every block transfer to the PrefetchBuffer. The kernel detects when the requested data resides in a new page. Before requesting the actual data, the kernel reads the data from the spare area section of the page, where the ECC parity data resides, to the PrefetchBuffer. Then the kernel stores the data internally on the stack to the EccParity structure.

The parity data for the entire NAND flash page is stored, allowing for error checking to be performed on all further data transfers from that page without requiring further access to the spare area. Thus the kernel adopts a more efficient access method when requesting the actual data, by only issuing a single read command for sequential 256 byte block accesses to a page.

Because the NAND flash boot procedure uses a prefetch mechanism, the 256 byte block following the end of the boot stream must have the correct ECC parity field programmed. Failure to adhere to this results in the boot kernel generating an uncorrectable error when fetching the block of data; and the boot process terminates.

Software Configurable NAND Flash Boot Modes

The NAND flash boot mode supports three different boot methods with regards to handling errors and bad blocks.

- Sequential Block Mode (default)
- Block Skip Mode
- Multiple Image Mode

The three booting options provide users with flexibility in how they use a NAND flash for booting purposes.

The three boot modes are configured through the `uwBlockSkipFeature` variable of the `EccParity` structure. By default `uwBlockSkipFeature` = 0, configuring the device for Sequential Block Mode. The user can change the boot mode by modifying the `uwBlockSkipFeature` variable in an initialization routine that is loaded and executed before the main application boot stream is processed. Access to the `ADI_BOOT_NAND` structure is provided by a pointer stored in the `dUserLong` parameter of the `ADI_BOOT_DATA` structure.

Sequential Block Mode

The default boot method is the Sequential Block Mode. In this mode no bad block detection is performed. The processor simply boots the boot stream starting from page 0 of block 0 until the end of the boot stream is reached. Error correction is always performed for greater reliability. However, if an uncorrectable error or error in the parity data is detected, the booting process terminates and the error handler is called.

This boot mode is suited for applications that wish to adopt a second stage boot loader approach, where the second stage loader starts from the first byte in the NAND flash.

If the boot stream to be loaded spans a number of blocks then all blocks that the boot stream occupies must be good blocks. If a block is known to be bad then this boot method should not be adopted for that particular device.

Figure 17-35 shows some typical usage scenarios for this mode.

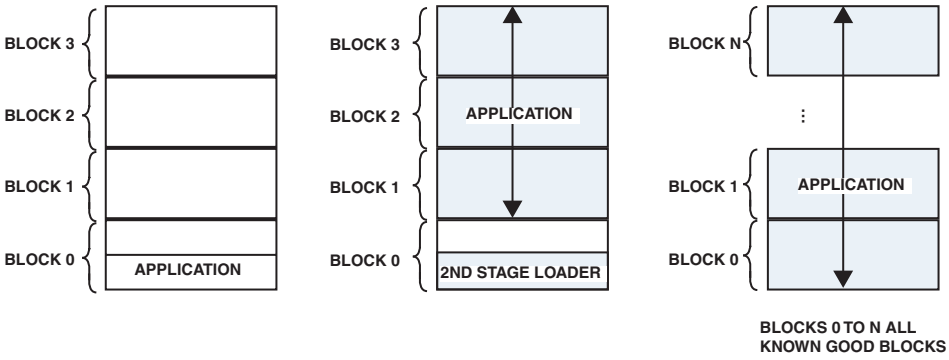


Figure 17-35. Sequential Block Mode Usage Scenarios

Block Skip Mode

This mode is enabled with `uwBlockSkipFeature` = 1. When enabling this mode the user must also set `uwBlockModifier` = 1. Failure to do so can result in the boot procedure failing.

This boot mode is suited for larger applications not adopting the second stage loader approach. During the loading of the application to the NAND flash, upon detection of factory set bad block, the last byte of the spare area of the first and second page of the bad block is set to a non 0xFF value.

The boot procedure works in a similar manner to the Sequential Block Mode except on detection of an access to a new block the spare area sections of the first two pages are loaded. The boot kernel checks the last byte of each. If either is not equal to 0xFF then the page is detected as bad. A byte offset of 1 block is then applied to all subsequent data requests thus skipping any bad blocks. This allows for the booting a single larger stream that is impeded by bad blocks in the area that the boot stream occupies.

Each time a bad block is encountered the byte offset applied to the address of the requested data is incremented by 1 block. Figure 17-36 highlights a typical usage scenario for this boot method.

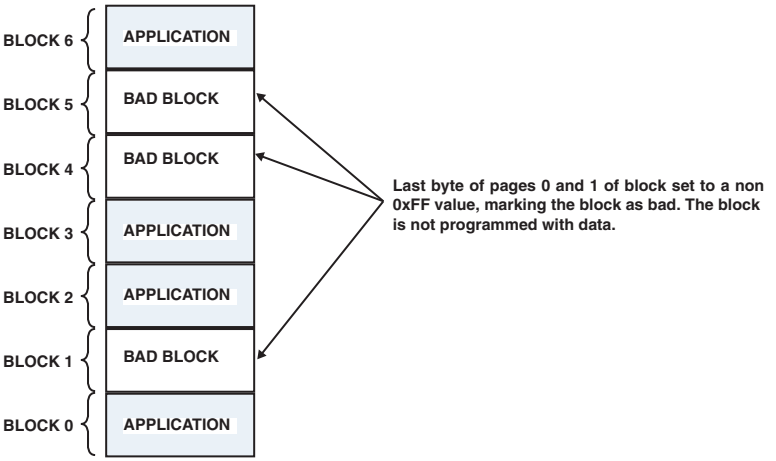


Figure 17-36. Block Skip Mode Typical Usage Scenario

Multiple Image Mode

This mode is enabled with `uwBlockSkipFeature` = 2. Multiple Image Mode allows for multiple copies of the boot stream to be loaded to the NAND flash providing maximum reliability. The number of blocks between each copy of the boot stream is defined by `uwBlockModifier`.

Upon detection of an access to a new block—as in Block Skip Mode, the last byte of the spare area of the first and second page of the block are checked to see if either indicate that the block is bad. If the block is bad, the block offset is applied to the requested data address to fetch from the next copy of the application.

This mode is the only mode that can handle uncorrectable errors from error detection and correction. If an uncorrectable error is received in any block (including block 0), or an error is detected in the parity data, the

kernel will fetch the same block of data from the next copy of the application.

The parameter `uwMaxCopies` specifies how many copies of the application are located in the NAND flash. If the processor is booting from the final copy and an uncorrectable error, error in the ECC parity data, or a bad block occurs —the processor enters a safe idle state and the booting process is terminated. This boot method provides greater reliability when regular boot stream updates are expected throughout the life of the product.

Figure 17-37 shows a typical usage scenario.

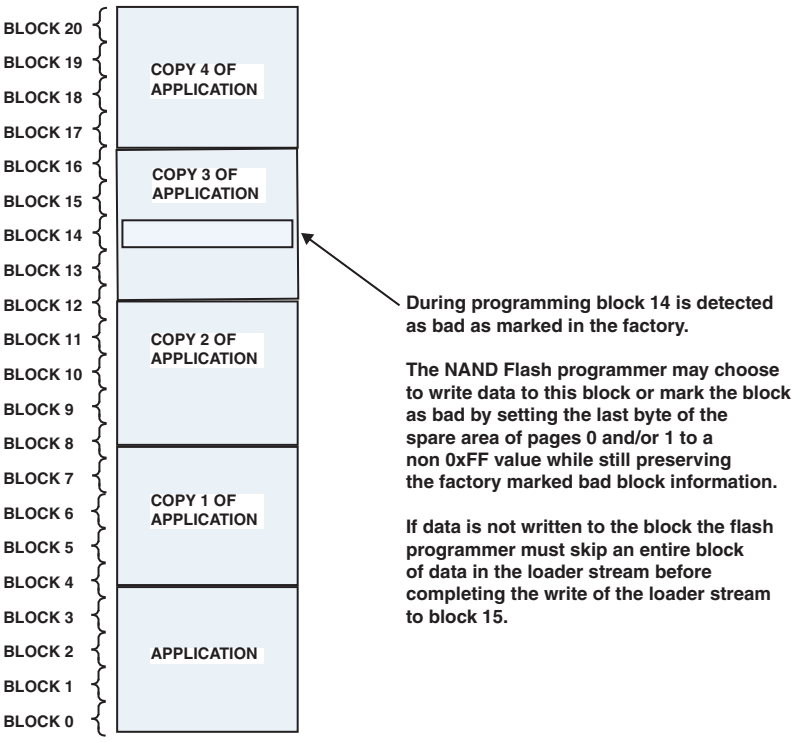

BLOCK 20
BLOCK 19
BLOCK 18
BLOCK 17
BLOCK 16
BLOCK 15
BLOCK 14
BLOCK 13
BLOCK 12
BLOCK 11
BLOCK 10
BLOCK 9
BLOCK 8
BLOCK 7
BLOCK 6
BLOCK 5
BLOCK 4
BLOCK 3
BLOCK 2
BLOCK 1
BLOCK 0
COPY 4 OF APPLICATION
COPY 3 OF APPLICATION
COPY 2 OF APPLICATION
COPY 1 OF APPLICATION
APPLICATION
During programming block 14 is detected as bad as marked in the factory.
The NAND Flash programmer may choose to write data to this block or mark the block as bad by setting the last byte of the spare area of pages 0 and/or 1 to a non 0xFF value while still preserving the factory marked bad block information.
If data is not written to the block the flash programmer must skip an entire block of data in the loader stream before completing the write of the loader stream to block 15.

Figure 17-37. Mirror Image Mode Typical Usage Scenario

Reset and Booting Registers

Two registers are used for reset and booting—the software reset register (SWRST) and the system reset configuration register (SYSCR).

Software Reset (SWRST) Register

A software reset can be initiated by setting bits [2:0] in the system software reset field in the software reset register (SWRST) shown in Figure 17-38 on page 17-100. Bit 3 can be used to generate reset upon core-double-fault. A core-double-fault resets both the core and the peripherals, but not the RTC block and most of the DPMC. Bit 15 indicates whether a software reset has occurred since the last time SWRST was read. Bit 14 indicates the software watchdog timer has generated the software reset. Bit 13 indicates the core-double-fault has generated the software reset. Bits [15:13] are read-only and cleared when the register is read. Reading the SWRST also clears bits [15:13] in the SYSCR register. Bits [3:0] are read/write.

Only writing to bits[2:0], resets only the modules in the SCLK domain. It does not clear the core. The program executes normally at the instruction after the MMR write to SWRST. The system is kept in the reset state as long as the bits[2:0] are set to b#111. To release reset, write a zero again. An example is shown in Listing 17-3 on page 17-139. It is not recommended

that this functionality be used directly. Rather, call the ROM function `bfrom_SysControl()` to perform a system reset.

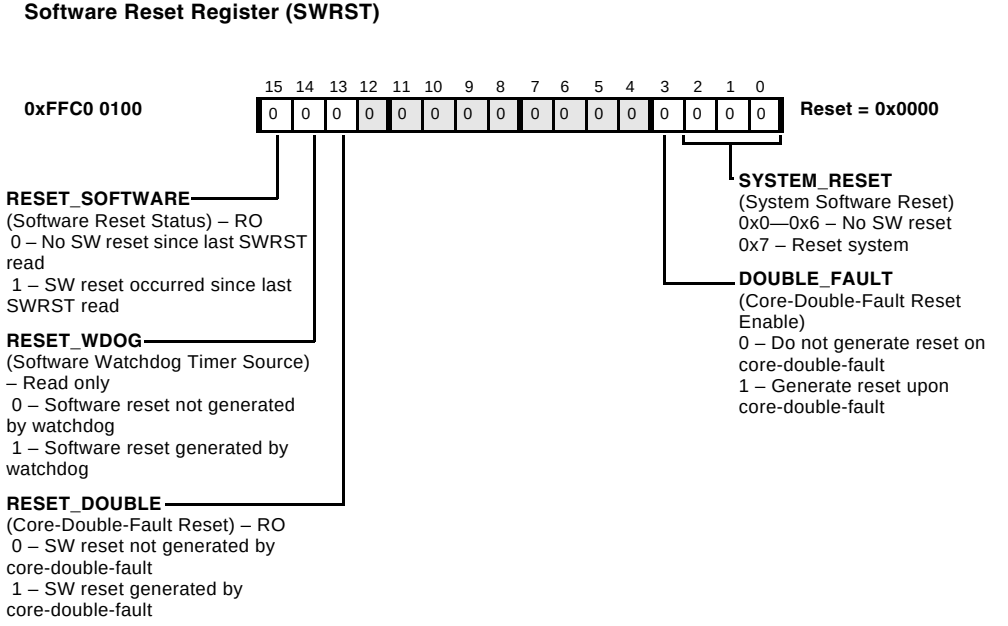

Software Reset Register (SWRST)
0xFFC0 0100
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RESET_SOFTWARE
(Software Reset Status) – RO
0 – No SW reset since last SWRST read
1 – SW reset occurred since last SWRST read
RESET_WDOG
(Software Watchdog Timer Source) – Read only
0 – Software reset not generated by watchdog
1 – Software reset generated by watchdog
RESET_DOUBLE
(Core-Double-Fault Reset) – RO
0 – SW reset not generated by core-double-fault
1 – SW reset generated by core-double-fault
SYSTEM_RESET
(System Software Reset)
0x0—0x6 – No SW reset
0x7 – Reset system
DOUBLE_FAULT
(Core-Double-Fault Reset Enable)
0 – Do not generate reset on core-double-fault
1 – Generate reset upon core-double-fault

Figure 17-38. Software Reset Register

System Reset Configuration (SYSCR) Register

The values sensed from the `BMODE[3:0]` pins are mirrored into the system reset configuration register (`SYSCR`). The values are available for software access and modification after the hardware reset sequence. Software can modify only bits[7:4] in this register to customize boot processing upon a software reset.

The bits [15:13] are exact copies of the same bits in the `SWRST` register. Unlike the `SWRST` register, `SYSCR` can be read without clearing these bits. Reading `SWRST` also causes `SYSCR[15:13]` to clear.

The WURESET indicates whether there was a wake up from hibernate since the last hardware reset. The bit cannot be cleared by software.

Bits [11:8] have no booting or reset purpose. These bits control the DMA arbitration.

The software reset configuration register (SYSCR) is shown in Figure 17-39 on page 17-102.

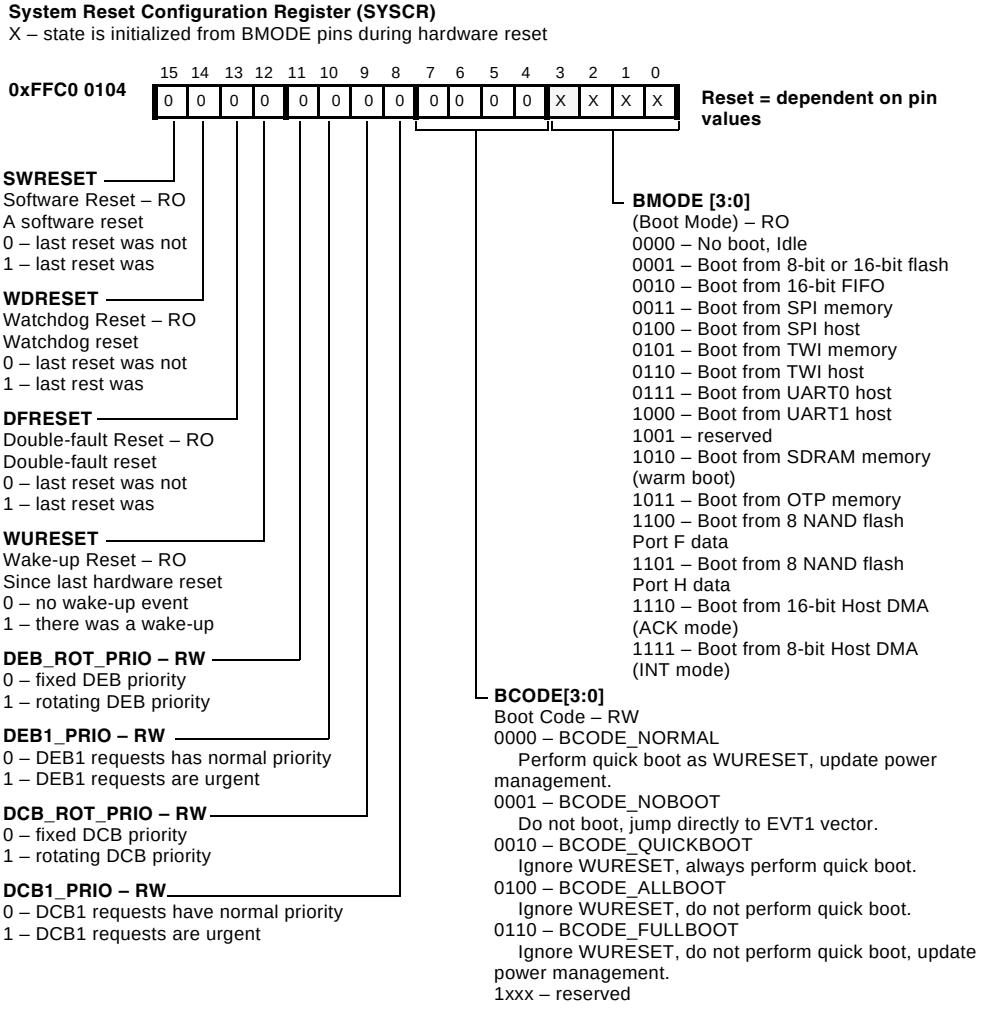

System Reset Configuration Register (SYSCR)
X – state is initialized from BMODE pins during hardware reset
0xFFC0 0104
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 X X X X
Reset = dependent on pin values
SWRESET
Software Reset – RO
A software reset
0 – last reset was not
1 – last reset was
WDRESET
Watchdog Reset – RO
Watchdog reset
0 – last reset was not
1 – last rest was
DFRESET
Double-fault Reset – RO
Double-fault reset
0 – last reset was not
1 – last reset was
WURESET
Wake-up Reset – RO
Since last hardware reset
0 – no wake-up event
1 – there was a wake-up
DEB_ROT_PRIO – RW
0 – fixed DEB priority
1 – rotating DEB priority
DEB1_PRIO – RW
0 – DEB1 requests has normal priority
1 – DEB1 requests are urgent
DCB_ROT_PRIO – RW
0 – fixed DCB priority
1 – rotating DCB priority
DCB1_PRIO – RW
0 – DCB1 requests have normal priority
1 – DCB1 requests are urgent
BMODE [3:0]
(Boot Mode) – RO
0000 – No boot, Idle
0001 – Boot from 8-bit or 16-bit flash
0010 – Boot from 16-bit FIFO
0011 – Boot from SPI memory
0100 – Boot from SPI host
0101 – Boot from TWI memory
0110 – Boot from TWI host
0111 – Boot from UART0 host
1000 – Boot from UART1 host
1001 – reserved
1010 – Boot from SDRAM memory (warm boot)
1011 – Boot from OTP memory
1100 – Boot from 8 NAND flash Port F data
1101 – Boot from 8 NAND flash Port H data
1110 – Boot from 16-bit Host DMA (ACK mode)
1111 – Boot from 8-bit Host DMA (INT mode)
BCODE[3:0]
Boot Code – RW
0000 – BCODE_NORMAL
Perform quick boot as WURESET, update power management.
0001 – BCODE_NOBOOT
Do not boot, jump directly to EVT1 vector.
0010 – BCODE_QUICKBOOT
Ignore WURESET, always perform quick boot.
0100 – BCODE_ALLBOOT
Ignore WURESET, do not perform quick boot.
0110 – BCODE_FULLBOOT
Ignore WURESET, do not perform quick boot, update power management.
1xxx – reserved

Figure 17-39. System Reset Configuration Register

Boot Code Revision Control (BK_REVISION)

The boot ROM reserves the 32-bits at address 0xEF00 0040 for a four byte version code as shown in Figure 17-40.

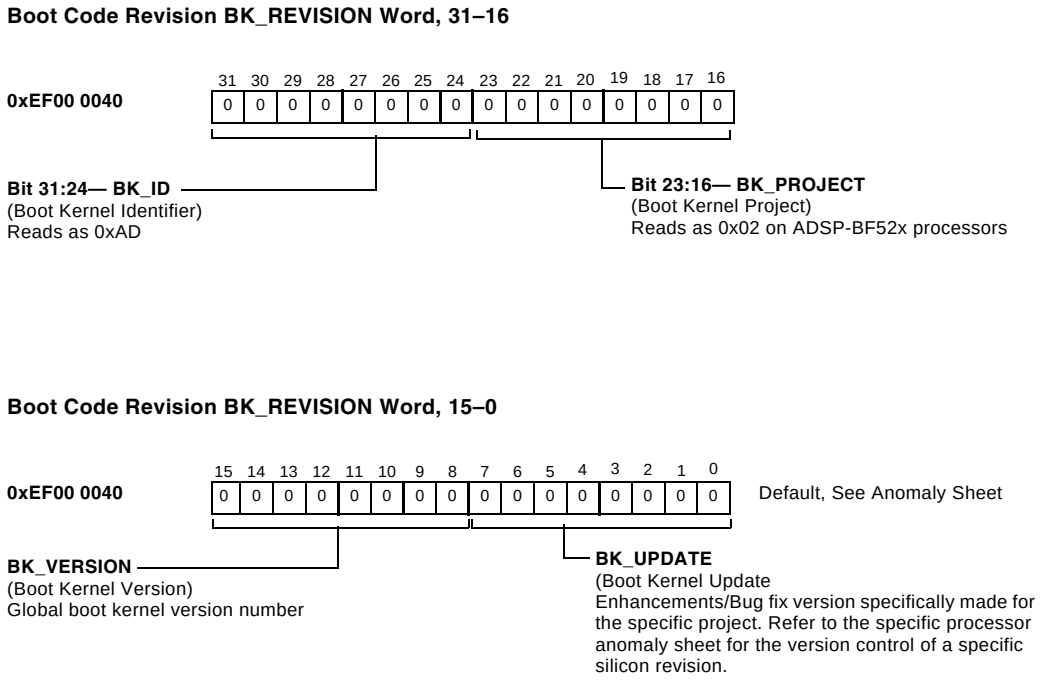

Boot Code Revision BK_REVISION Word, 31–16
0xEF00 0040
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Bit 31:24— BK_ID
(Boot Kernel Identifier)
Reads as 0xAD
Bit 23:16— BK_PROJECT
(Boot Kernel Project)
Reads as 0x02 on ADSP-BF52x processors
Boot Code Revision BK_REVISION Word, 15–0
0xEF00 0040
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default, See Anomaly Sheet
BK_VERSION
(Boot Kernel Version)
Global boot kernel version number
BK_UPDATE
(Boot Kernel Update
Enhancements/Bug fix version specifically made for the specific project. Refer to the specific processor anomaly sheet for the version control of a specific silicon revision.

Figure 17-40. Boot Code Revision Code (BK_REVISION)

Boot Code Date Code (BK_DATECODE)

The boot ROM reserves the 32-bits at address 0xEF00 0050 for the build date as shown in Figure 17-41.

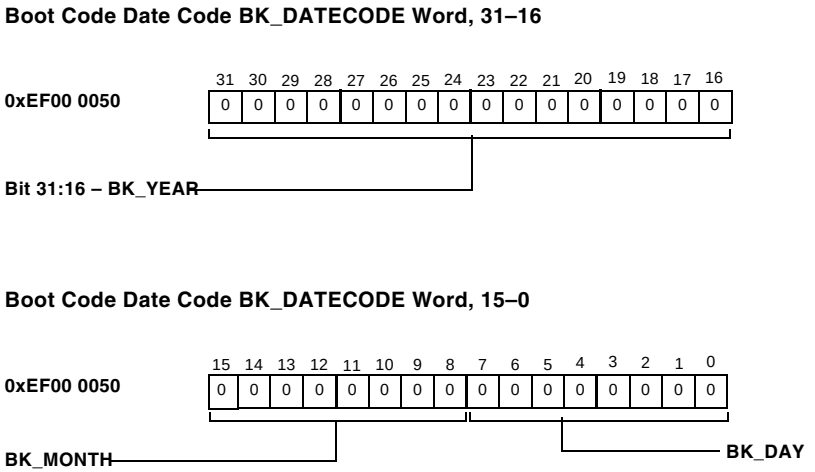


Figure 17-41. Boot Code Date Code (BK_DATECODE)

Zero Word (BK_ZEROS)

The boot ROM reserves the 32-bits at address 0xEF00 0048 which always reads as 0x0000 000 as shown in Figure 17-42.

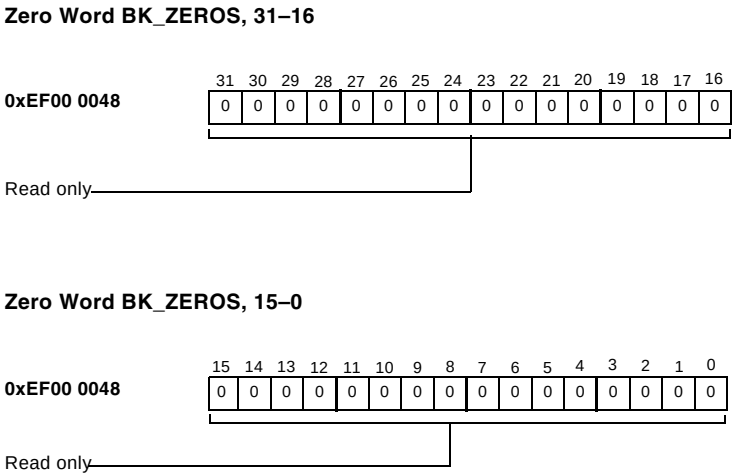


Figure 17-42. Zero Word (BK_ZEROS)

Ones Word (BK_ONES)

The boot ROM reserves the 32-bits at address 0xEF00 004C which always reads 0xFFFF FFFF as shown in Figure 17-43.

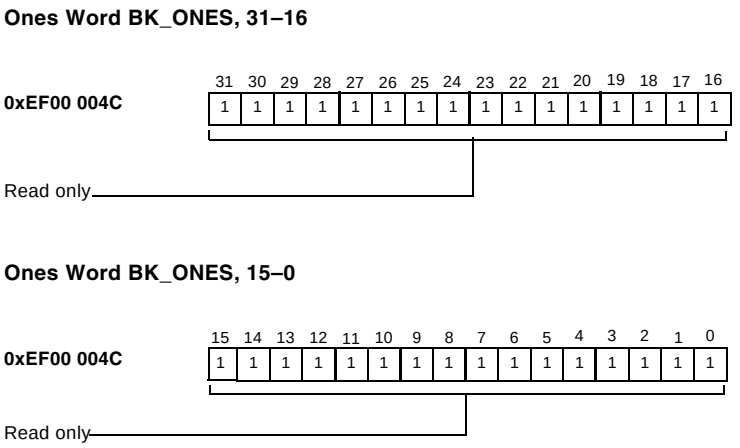


Figure 17-43. Ones Word (BK_ONES)

OTP Memory Pages for Booting

Lower PBS00 Half Page

The 64-bit lower half of page 0x18 is always read by the preboot routine. These control bits customize the boot process and instruct the preboot routine whether to process further pages and whether the PLL settings

have to be changed. Other bits customize the SPI and TWI master boot speed.

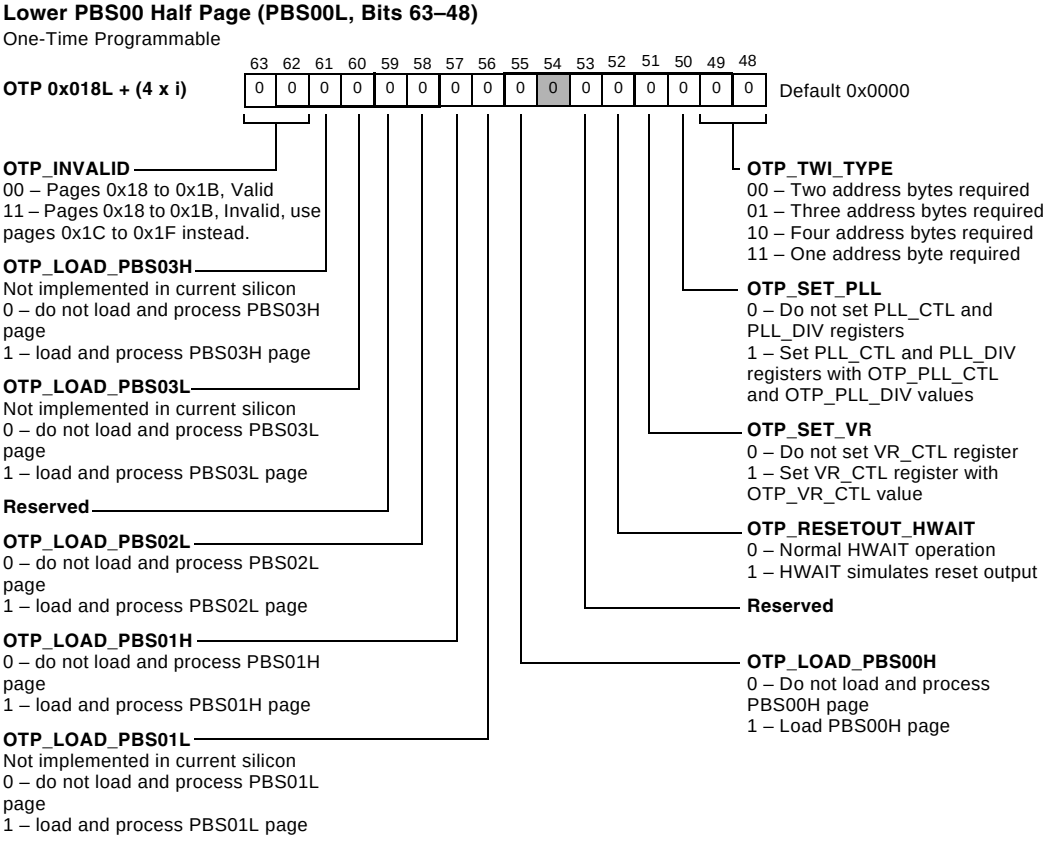

Lower PBS00 Half Page (PBS00L, Bits 63–48)
One-Time Programmable
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48
OTP 0x018L + (4 x i)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
OTP_INVALID
00 – Pages 0x18 to 0x1B, Valid
11 – Pages 0x18 to 0x1B, Invalid, use pages 0x1C to 0x1F instead.
OTP_LOAD_PBS03H
Not implemented in current silicon
0 – do not load and process PBS03H page
1 – load and process PBS03H page
OTP_LOAD_PBS03L
Not implemented in current silicon
0 – do not load and process PBS03L page
1 – load and process PBS03L page
Reserved
OTP_LOAD_PBS02L
0 – do not load and process PBS02L page
1 – load and process PBS02L page
OTP_LOAD_PBS01H
0 – do not load and process PBS01H page
1 – load and process PBS01H page
OTP_LOAD_PBS01L
Not implemented in current silicon
0 – do not load and process PBS01L page
1 – load and process PBS01L page
OTP_TWI_TYPE
00 – Two address bytes required
01 – Three address bytes required
10 – Four address bytes required
11 – One address byte required
OTP_SET_PLL
0 – Do not set PLL_CTL and PLL_DIV registers
1 – Set PLL_CTL and PLL_DIV registers with OTP_PLL_CTL and OTP_PLL_DIV values
OTP_SET_VR
0 – Do not set VR_CTL register
1 – Set VR_CTL register with OTP_VR_CTL value
OTP_RESETOUT_HWAIT
0 – Normal HWAIT operation
1 – HWAIT simulates reset output
Reserved
OTP_LOAD_PBS00H
0 – Do not load and process PBS00H page
1 – Load PBS00H page

Figure 17-44. Lower PBS00 Half Page (PBS00L, Bits 63–48)

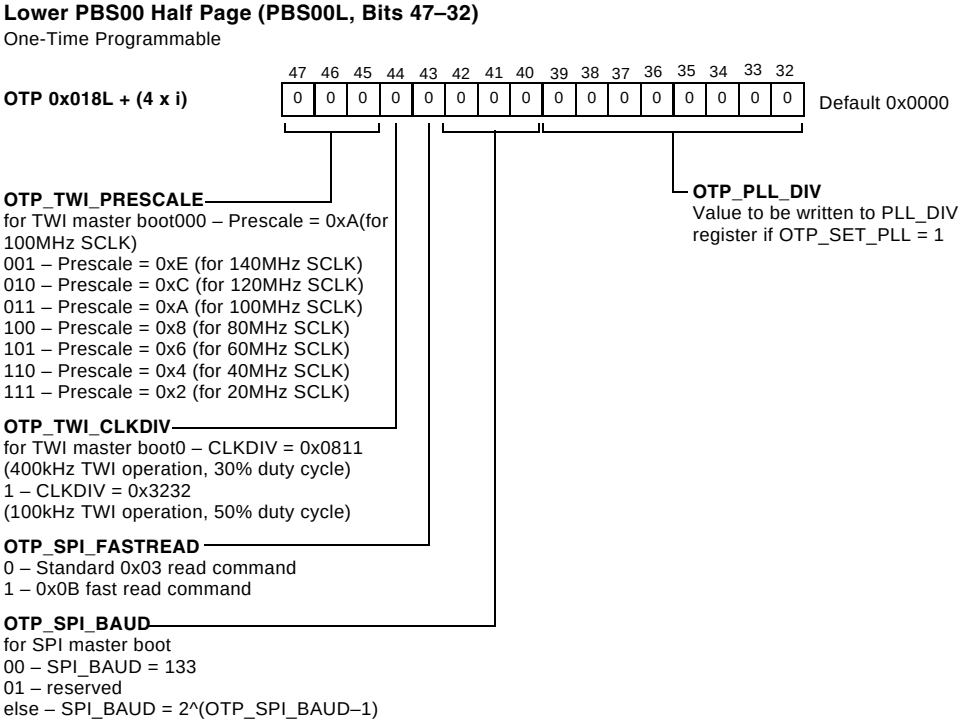

Lower PBS00 Half Page (PBS00L, Bits 47–32)
One-Time Programmable
47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
OTP 0x018L + (4 x i)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
OTP_TWI_PRESCALE
for TWI master boot000 – Prescale = 0xA(for 100MHz SCLK)
001 – Prescale = 0xE (for 140MHz SCLK)
010 – Prescale = 0xC (for 120MHz SCLK)
011 – Prescale = 0xA (for 100MHz SCLK)
100 – Prescale = 0x8 (for 80MHz SCLK)
101 – Prescale = 0x6 (for 60MHz SCLK)
110 – Prescale = 0x4 (for 40MHz SCLK)
111 – Prescale = 0x2 (for 20MHz SCLK)
OTP_TWI_CLKDIV
for TWI master boot0 – CLKDIV = 0x0811
(400kHz TWI operation, 30% duty cycle)
1 – CLKDIV = 0x3232
(100kHz TWI operation, 50% duty cycle)
OTP_SPI_FASTREAD
0 – Standard 0x03 read command
1 – 0x0B fast read command
OTP_SPI_BAUD
for SPI master boot
00 – SPI_BAUD = 133
01 – reserved
else – SPI_BAUD = 2^(OTP_SPI_BAUD–1)
OTP_PLL_DIV
Value to be written to PLL_DIV register if OTP_SET_PLL = 1

Figure 17-45. Lower PBS00 Half Page (PBS00L, Bits 47–32)

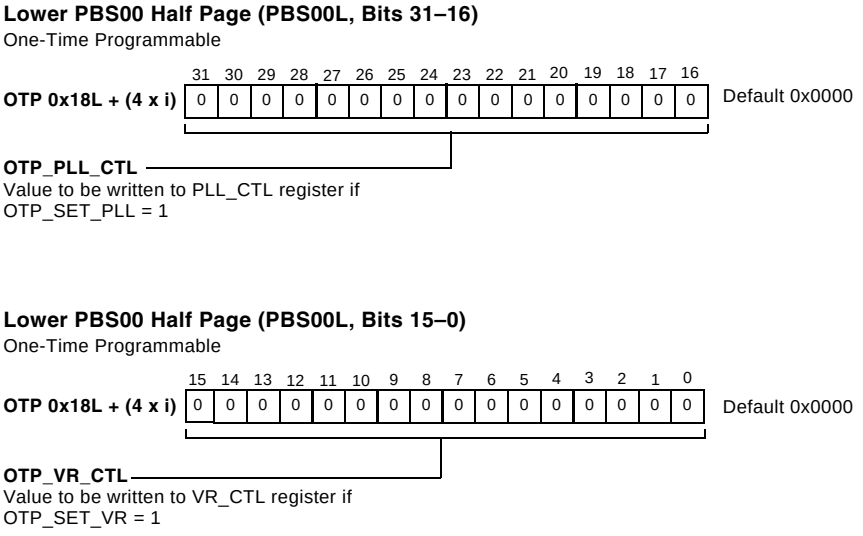

Lower PBS00 Half Page (PBS00L, Bits 31–16)
One-Time Programmable
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
OTP 0x18L + (4 x i)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
OTP_PLL_CTL
Value to be written to PLL_CTL register if
OTP_SET_PLL = 1
Lower PBS00 Half Page (PBS00L, Bits 15–0)
One-Time Programmable
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OTP 0x18L + (4 x i)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
OTP_VR_CTL
Value to be written to VR_CTL register if
OTP_SET_VR = 1

Figure 17-46. Lower PBS00 Half Page (PBS00L, Bits 31–0)

Upper PBS00 Half Page

The preboot routine loads the upper 64-bit half of page PBS00 only if the OTP_LOAD_PBS00H bit in the PBS00L page is set. Page PBS00H customizes the default setting of the asynchronous portion of the EBIU controller.

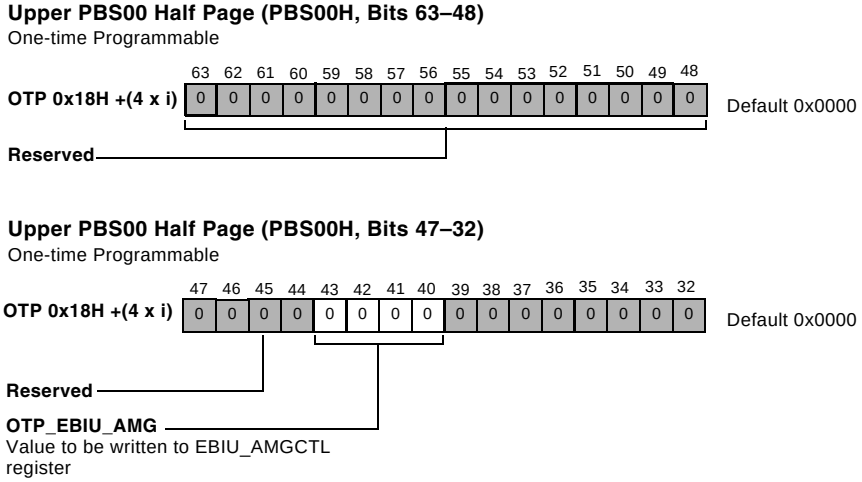

Upper PBS00 Half Page (PBS00H, Bits 63–48)
One-time Programmable
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48
OTP 0x18H +(4 x i)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
Reserved
Upper PBS00 Half Page (PBS00H, Bits 47–32)
One-time Programmable
47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
OTP 0x18H +(4 x i)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
Reserved
OTP_EBIU_AMG
Value to be written to EBIU_AMGCTL register

Figure 17-47. Upper PBS00 Half Page (PBS00H, Bits 63–32)

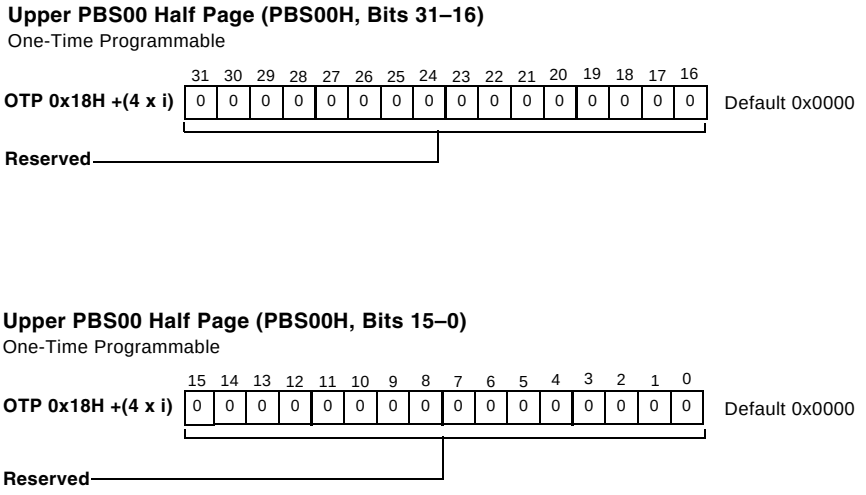

Upper PBS00 Half Page (PBS00H, Bits 31–16)
One-Time Programmable
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
OTP 0x18H +(4 x i)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
Reserved
Upper PBS00 Half Page (PBS00H, Bits 15–0)
One-Time Programmable
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OTP 0x18H +(4 x i)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
Reserved

Figure 17-48. Upper PBS00 Half Page (PBS00H, Bits 31–0)

Lower PBS01 Half Page

The half page `PBS01L` is reserved and not used in the current silicon.

Do not use this page as it may be populated in future silicon revisions.

Upper PBS01 Half Page

The preboot routine loads the upper 64-bit half of page 0x19 only if the `OTP_LOAD_PBS01H` bit in the `PBS00L` page is set. This page allows the user to disable boot modes. If a disabled boot mode configuration is chosen by the `BMODE[3:0]` pins, the boot kernel goes into idle state. This half page also provides customization of the NAND flash controller. In OTP boot

mode, this pages determines where in OTP memory the boot stream resides.

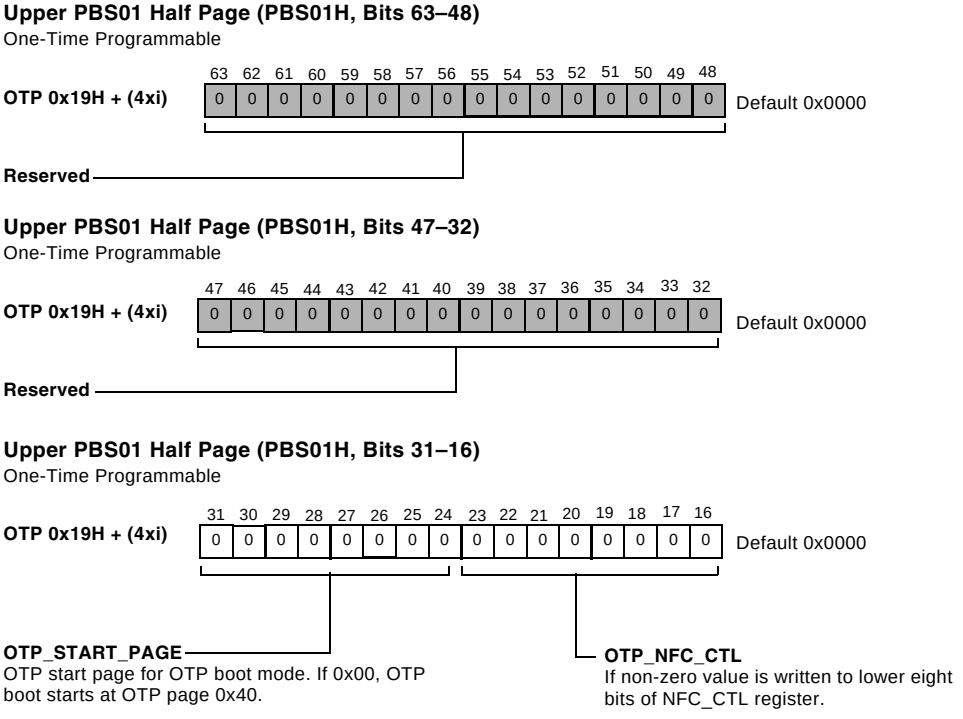

Upper PBS01 Half Page (PBS01H, Bits 63–48)
One-Time Programmable
OTP 0x19H + (4xi)
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
Reserved
Upper PBS01 Half Page (PBS01H, Bits 47–32)
One-Time Programmable
OTP 0x19H + (4xi)
47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
Reserved
Upper PBS01 Half Page (PBS01H, Bits 31–16)
One-Time Programmable
OTP 0x19H + (4xi)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
OTP_START_PAGE
OTP start page for OTP boot mode. If 0x00, OTP boot starts at OTP page 0x40.
OTP_NFC_CTL
If non-zero value is written to lower eight bits of NFC_CTL register.

Figure 17-49. OTP Half Page (PBS01H, Bits 63–16)

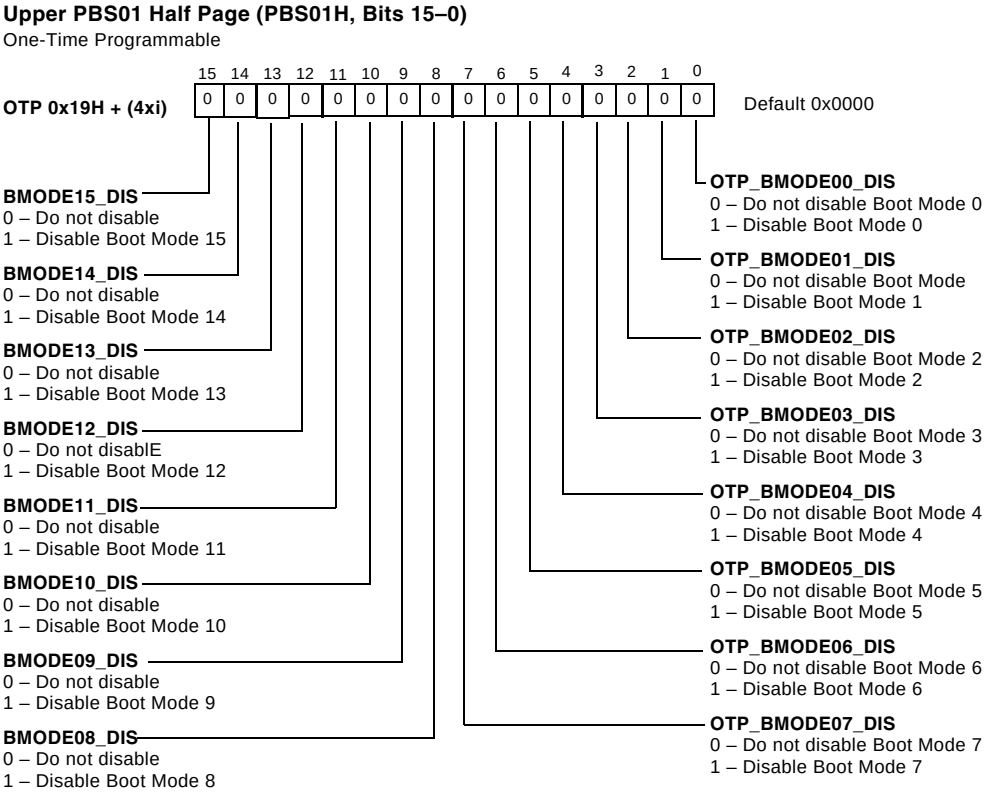

Upper PBS01 Half Page (PBS01H, Bits 15–0)
One-Time Programmable
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OTP 0x19H + (4xi)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
BMODE15_DIS
0 – Do not disable
1 – Disable Boot Mode 15
BMODE14_DIS
0 – Do not disable
1 – Disable Boot Mode 14
BMODE13_DIS
0 – Do not disable
1 – Disable Boot Mode 13
BMODE12_DIS
0 – Do not disablE
1 – Disable Boot Mode 12
BMODE11_DIS
0 – Do not disable
1 – Disable Boot Mode 11
BMODE10_DIS
0 – Do not disable
1 – Disable Boot Mode 10
BMODE09_DIS
0 – Do not disable
1 – Disable Boot Mode 9
BMODE08_DIS
0 – Do not disable
1 – Disable Boot Mode 8
OTP_BMODE00_DIS
0 – Do not disable Boot Mode 0
1 – Disable Boot Mode 0
OTP_BMODE01_DIS
0 – Do not disable Boot Mode
1 – Disable Boot Mode 1
OTP_BMODE02_DIS
0 – Do not disable Boot Mode 2
1 – Disable Boot Mode 2
OTP_BMODE03_DIS
0 – Do not disable Boot Mode 3
1 – Disable Boot Mode 3
OTP_BMODE04_DIS
0 – Do not disable Boot Mode 4
1 – Disable Boot Mode 4
OTP_BMODE05_DIS
0 – Do not disable Boot Mode 5
1 – Disable Boot Mode 5
OTP_BMODE06_DIS
0 – Do not disable Boot Mode 6
1 – Disable Boot Mode 6
OTP_BMODE07_DIS
0 – Do not disable Boot Mode 7
1 – Disable Boot Mode 7

Figure 17-50. OTP Half Page PBS01H (PBS01H, Bits 15–0)

Lower PBS02 Half Page

The preboot routine loads the lower 64-bit half of page 0x1A only if the OTP_LOAD_PBS02L bit in half page PBS00L is set. Half pages PBS02L and PBS02H customize the SDRAM controller settings.

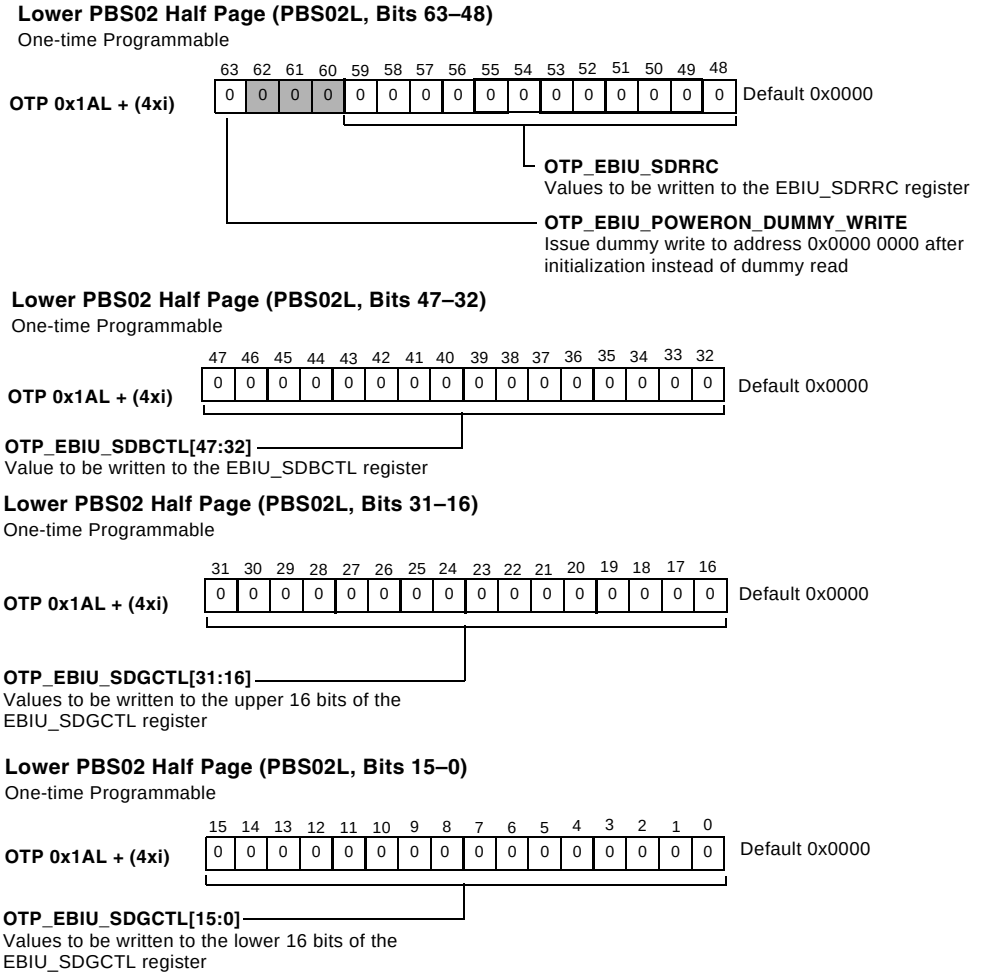

Lower PBS02 Half Page (PBS02L, Bits 63–48)
One-time Programmable
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48
OTP 0x1AL + (4xi)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
OTP_EBIU_SDRRC
Values to be written to the EBIU_SDRRC register
OTP_EBIU_POWERON_DUMMY_WRITE
Issue dummy write to address 0x0000 0000 after initialization instead of dummy read
Lower PBS02 Half Page (PBS02L, Bits 47–32)
One-time Programmable
47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
OTP 0x1AL + (4xi)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
OTP_EBIU_SDBCTL[47:32]
Value to be written to the EBIU_SDBCTL register
Lower PBS02 Half Page (PBS02L, Bits 31–16)
One-time Programmable
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
OTP 0x1AL + (4xi)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
OTP_EBIU_SDGCTL[31:16]
Values to be written to the upper 16 bits of the EBIU_SDGCTL register
Lower PBS02 Half Page (PBS02L, Bits 15–0)
One-time Programmable
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OTP 0x1AL + (4xi)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Default 0x0000
OTP_EBIU_SDGCTL[15:0]
Values to be written to the lower 16 bits of the EBIU_SDGCTL register

Figure 17-51. Lower PBS02 Half Page (PBS02L Bits 63–0)

Upper PBS02 Half Page

Page PBS02H is reserved. Do not use for any purpose.

Reserved Half Pages

The half pages PBS01L, PBS02H, PBS03L and PBS03H are reserved and not used in the current silicon.

Do not use these pages as they may be populated in future silicon revisions.

Data Structures

The boot kernel uses specific data structures for internal processing. Advanced users can customize the booting process by changing the content of the structure within the initcode routines. This section uses C language definitions for documentation purposes. VisualDSP++ users can use these structures directly in assembly programs by using the .IMPORT directive. The structures are supplied by the bfrom.h header file in your VisualDSP++ installation directory.

ADI_BOOT_HEADER

```
typedef struct   {
   s32  dBlockCode;
   void* pTargetAddress;
   s32  dByteCount;
   s32  dArgument;
}   ADI_BOOT_HEADER;
```

The structure ADI_BOOT_HEADER is used by the boot kernel to load and process a block header.

Every block header is loaded to L1 data memory location 0xFF80 7FF0–0xFF80 7FFF first or where `pHeader` points to. There it is analyzed by the boot kernel.

ADI_BOOT_BUFFER

```
typedef struct   {
   void*  pSource;
   s32    dByteCount;
}   ADI_BOOT_BUFFER;
```

The structure `ADI_BOOT_BUFFER` is used for any kind of buffer. For the user, this structure is important when implementing advanced callback mechanisms.

ADI_BOOT_DATA

```
typedef struct {
   void*  pSource;
   void*  pDestination;
   s16*   pControlRegister;
   s16*   pDmaControlRegister;
   s32    dControlValue;
   s32    dByteCount;
   s32    dFlags;
   s16    uwDataWidth;
   s16    uwSrcModifyMult;
   s16    uwDstModifyMult;
   s16    uwHwait;
   s16    uwSsel;
   s16    uwUserShort;
   s32    dUserLong;
   s32    dReserved;
   ADI_BOOT_ERROR_FUNC*  pErrorFunction;
   ADI_BOOT_LOAD_FUNC*  pLoadFunction;
   ADI_BOOT_CALLBACK_FUNC*  pCallBackFunction;
```

```
    ADI_BOOT_HEADER* pHeader;
    void*  pTempBuffer;
    void*  pTempCurrent;
    s32    dTempByteCount;
    s32    dBlockCount;
    s32    dClock;
    void*  pLogBuffer;
    void*  pLogCurrent;
    s32    dLogByteCount;
}   ADI_BOOT_DATA;
```

The structure `ADI_BOOT_DATA` is the main data structure. A pointer to a `ADI_BOOT_DATA` structure is passed to most complex subroutines, including load functions, initcode, and callback routines. The structure has two parts. While the first is closely related to internal memory load routines, the second provides access to global boot settings.

Table 17-16 on page 17-117 describes the data structures.

Table 17-16. Structure Variables, `ADI_BOOT_DATA`

Variable	Description
`pSource`	In the context of the boot kernel, the `pSource` pointer points either to the start address of the entire boot stream or to the header of the next boot block. In the context of memory load routines `pSource` points to the source address of the DMA work unit.
`pDestination`	The `pDestination` pointer is only used in memory load routines. It points to the destination address of the DMA work unit. It points to either 0xFF80 7FF0 when a header is loaded, or the target address when the payload data is loaded.
`pControlRegister`	This pointer holds the MMR address of the peripheral's main control register (for example `UARTx_LCR` or `SPIx_CTL`)
`pDmaControlRegister`	This pointer holds the MMR address of the `DMAx_CONFIG` register for the DMA channel in use.
`dControlValue`	The lower 16 bits of this value are written to the `pControlRegister` location each time a DMA work unit is started.

Table 17-16. Structure Variables, ADI_BOOT_DATA (Continued)

Variable	Description
dByteCount	Number of bytes to be transferred.
dFlags	The lower 16 bits of this variable hold the lower 16 bits of the current block code. The upper 16 bits hold global flags. See “dFlags Word” on page 17-120.
uwDataWidth	This instructs the memory load routine to use: 0 – 8-bit DMA 1 – 16-bit DMA 2 – 32-bit DMA
uwSrcModifyMult	This is the multiplication factor used by the DMA source. A value of 1 sets the source modifier to 1 for 8-bit DMA, 2 for 16-bit DMA, or 4 for 32-bit DMA.
uwDstModifyMult	This is the multiplication factor used by the DMA destination. A value of 1 sets the destination modifier to 1 for 8-bit DMA, 2 for 16-bit DMA, or 4 for 32-bit DMA.
uwHwait	This 16-bit value holds the GPIO used for HWAIT signaling. The value can change on the fly. The upper eight bits designate the port number (for example 01 for Port A, 02 for Port B). The lower four bits designate the GPIO in the port. For example, GPIO PG0 has a value of 0x0700.
uwSsel	This 16-bit value holds the GPIO used for SPI slave select. The value can change on the fly. The upper eight bits designate the port number (for example 01 for Port A, 02 for Port B). The lower four bits designate the GPIO in the port.
uwUserShort	The programmer can use this 16-bit value for passing parameters between modules of a customized booting scheme.
dUserLong	The programmer can use this 32-bit value for passing parameters between modules of a customized booting scheme.
dReserved	This 32-bit value is reserved for future development.
pErrorFunction	This is the pointer to the error handler. See “Error Handler” on page 17-46.
pLoadFunction	This is the pointer to the function responsible for loading data. See “Load Functions” on page 17-47
pCallBackFunction;	This is the pointer to the callback function. See “Callback Routines” on page 17-43

Table 17-16. Structure Variables, ADI_BOOT_DATA (Continued)

Variable	Description
pHeader	The pHeader pointer holds the address for intermediate storage of the block header. By default this value is set to 0xFF80 7FF0.
pTempBuffer	This pointer tells the boot kernel what memory to use for intermediate storage when the BFLAG_INDIRECT flag is set for a given block. The pointer defaults to 0xFF90 7E00. The value can be modified by the initcode routine, but there would be some impact to the VisualDSP++ tools.
pTempCurrent	Defaults to the pTempBuffer value. A load function can modify this value to manipulate subsequent callback and memory DMA routines.
dTempByteCount	This is the size of the intermediate storage buffer used when the BFLAG_INDIRECT flag is set for a given block. This value defaults to 256 and can be modified by an initcode routine. When increasing this value, the pTempBuffer must also be changed since by default the block is at the end of a physical data memory block.
dBlockCount	This 32-bit variable counts the boot blocks that are processed by the boot kernel. If the user sets this value to a negative value, the boot kernel exits when the variable increments to zero.
dClock	The dClock variable holds information about the clock divider used by individual (serial) boot modes.
pLogBuffer	Pointer to the circular log buffer. By default the log buffer resides in L1 scratch pad memory at address 0xFFB0 0400.
pLogCurrent	Pointer to the next free entry of the circular log buffer.
dLogByteCount	Size of the circular log buffer, default is 0x400 bytes.

dFlags Word

Figure 17-53 and Figure 17-52 on page 17-120 describe the `dFlags` word. `dFlags [15-0]` is a copy of Block Code[15–0] of the block currently being processed.

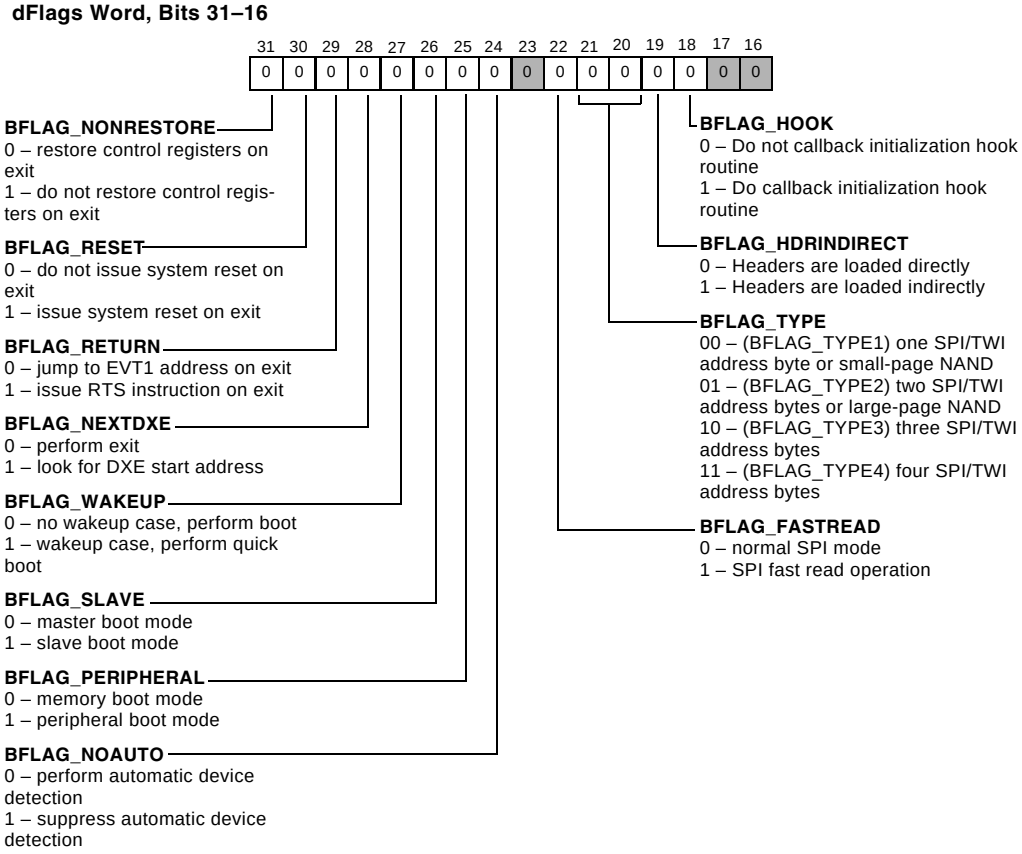

dFlags Word, Bits 31–16
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
BFLAG_NONRESTORE
0 – restore control registers on exit
1 – do not restore control registers on exit
BFLAG_RESET
0 – do not issue system reset on exit
1 – issue system reset on exit
BFLAG_RETURN
0 – jump to EVT1 address on exit
1 – issue RTS instruction on exit
BFLAG_NEXTDXE
0 – perform exit
1 – look for DXE start address
BFLAG_WAKEUP
0 – no wakeup case, perform boot
1 – wakeup case, perform quick boot
BFLAG_SLAVE
0 – master boot mode
1 – slave boot mode
BFLAG_PERIPHERAL
0 – memory boot mode
1 – peripheral boot mode
BFLAG_NOAUTO
0 – perform automatic device detection
1 – suppress automatic device detection
BFLAG_HOOK
0 – Do not callback initialization hook routine
1 – Do callback initialization hook routine
BFLAG_HDRINDIRECT
0 – Headers are loaded directly
1 – Headers are loaded indirectly
BFLAG_TYPE
00 – (BFLAG_TYPE1) one SPI/TWI address byte or small-page NAND
01 – (BFLAG_TYPE2) two SPI/TWI address bytes or large-page NAND
10 – (BFLAG_TYPE3) three SPI/TWI address bytes
11 – (BFLAG_TYPE4) four SPI/TWI address bytes
BFLAG_FASTREAD
0 – normal SPI mode
1 – SPI fast read operation

Figure 17-52. dFlags Word (Bits 31–16)

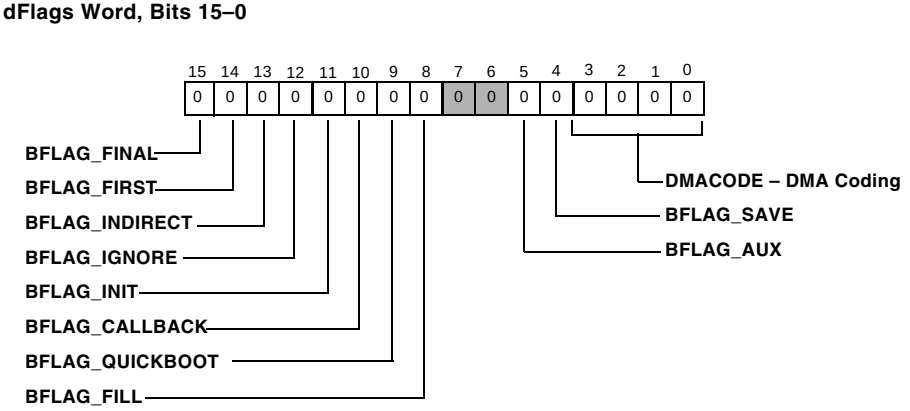


Figure 17-53. dFlags Word (Bits 15–0)

ADI_BOOT_NAND

```
typedef struct   {
   ADI_BOOT_NAND_DEVICE DeviceInfo;
   ADI_BOOT_NAND_BUFFER MainBuffer;
   ADI_BOOT_NAND_BUFFER PrefetchBuffer;
   ADI_BOOT_NAND_ACCESS AddressRequested;
   ADI_BOOT_NAND_ADDRESS AddressCycles;
   ADI_BOOT_NAND_ECC EccParity;
   ADI_BOOT_DATA *pBootData;
   void *pReserved;
}   ADI_BOOT_NAND;
```

The boot kernel uses a number of data structures for internal processing. Advanced users may manipulate some of contents of the structures from within initcode routines to customize the boot process further.

`ADI_BOOT_NAND` is the central structure and is used only by the NAND flash boot kernel. The pointer to `ADI_BOOT_NAND` is stored in the `dUserLong` parameter of `ADI_BOOT_DATA` when NAND flash boot mode is enabled.

This pointer provides access to the ADI_BOOT_NAND structure through initialization routines to further customize the booting process.

Table 17-17. Structure Variables, ADI_BOOT_NAND

Variable	Description
DeviceInfo	Properties relating to the NAND flash device
MainBuffer	Information relating to the current contents of the MainBuffer.
PrefetchBuffer	Information relating to the current contents of the PrefetchBuffer.
AddressRequested	Details of the requested address when the address is converted to an address suitable for accessing the NAND flash.
AddressCycles	Information required to correctly read from the NAND flash device.
EccParity	Stores the error correction parity data for a NAND flash page and controls the operating mode of the NAND flash boot kernel.
pBootData	Pointer to the global ADI_BOOT_DATA structure.
pReserved	Reserved for future enhancements. Do not use.

ADI_BOOT_NAND_DEVICE

```
typedef struct   {
   u32  udIdCode;
   u32  udIdType;
   u16  uwBusWidth;
   u16  uwColumnMaskCount;
   u32  udColumnMask;
   u16  uwPageMaskCount;
   u32  udPageMask;
   u16  uwSpareMaskCount;
   u16  uwSpareAreaBit;
   u32  udBlockSize;
   u16  uwPageSize;
   u16  uwPagesPerBlock;
   u16  uwSpareAreaSize;
   u16  uwSpareAreaModifier;
```

```
    u16  uwNFCPages;
}   ADI_BOOT_NAND_DEVICE;
```

This structure provides details about the NAND flash device connected to the NFC. For booting from supported small-page NAND flash devices not all parameters are used and thus initialized. For supported large-page NAND flash memories, the structure is initialized after reading the electronic signature of the device. The fourth byte of the four byte electronic signature contains information for initialization of the entire structure.

Table 17-18. Structure Variables, `ADI_BOOT_NAND_DEVICE`

Variable	Description
`udIdCode`	The electronic signature of the device as received after issuing the `Read Electronic Signature` command. This is only used for large-page NAND flash devices. It is not populated if a small-page device is detected, since only a single small-page type is supported.
`udType`	0 indicates a small-page device. 1 indicates a large-page device.
`uwBusWidth`	Bus width of the device. '0' for 8-bit.
`uwColumnMaskCount`	Number of bits required to address all columns within a NAND flash page (excluding the spare area). This is used to translate the address `pSource` in `ADI_BOOT_DATA` to the format required for addressing the NAND flash device.
`udColumnMask`	Used to extract the column within a page being addressed from the requested source address.
`uwPageMaskCount`	Number of bits required to address all pages within a single NAND flash block.
`udPageMask`	Used to extract the page number within a block being addressed from the source address.
`uwSpareMaskCount`	Number of bits required to address all columns within the spare area at the end of a NAND flash page.
`uwSpareAreaBit`	Contains the bit position to be set to address the spare area of the NAND flash page.
`udBlockSize`	Block size of the device in bytes (excluding the spare area).
`uwPageSize`	Page size of the device in bytes (excluding the spare area).

Table 17-18. Structure Variables, ADI_BOOT_NAND_DEVICE (Continued)

Variable	Description
uwPagesPerBlock	Number of pages within a block.
uwSpareAreaSize	Number of bytes within the spare area of a page.
uwSpareAreaModifier	Number of bytes in the spare area dedicated to each 256 byte NAND flash controller page.
uwNFCPages	Number of 256 byte NAND Flash controller pages within a full NAND flash page.

ADI_BOOT_NAND_BUFFER

```
typedef struct   {
   void * pBegin;
   u16  uwLoadedNFCPage;
   u16  uwLoadedNANDPage;
   u16  uwLoadedNANDBlock;
}   ADI_BOOT_NAND_BUFFER;
```

The ADI_BOOT_NAND_BUFFER structure provides details of the current contents of a 256 byte buffer. There are two of these buffers required for NAND flash boot. The buffer provides details on the location of the buffer as well as its current contents. Since 256 byte blocks of data are read from the NAND flash memory at a time, the kernel can determine if a new data fetch is required from the NAND flash or whether the data resides in one of the two buffers located in internal memory.

Table 17-19. Structure Variables, ADI_BOOT_NAND_BUFFER

Variable	Description
pBegin	Pointer to the first address of a 256 byte buffer.
uwnLoadedNFCPage	The currently loaded 256 byte NAND flash controller sub-page.
uwLoadedNANDPage	The currently loaded NAND flash page.
uwLoadedNANDBlock	The currently loaded NAND flash block.

ADI_BOOT_NAND_ACCESS

```
typedef struct   {
   u16  uwAccessNFCPage;
   u16  uwAccessNANDPage;
   u16  uwAccessNANDBlock;
}   ADI_BOOT_NAND_ACCESS;
```

The actual page and block in which the data resides can be calculated from the source address provided by the main kernel and the contents of the ADI_BOOT_NAND_DEVICE structure. This structure is also used along with the ADI_BOOT_NAND_BUFFER to determine if data needs to be fetched from the NAND flash memory or whether it already resides in internal memory.

Table 17-20. Structure Variables, ADI_BOOT_NAND_ACCESS

Variable	Description
uwAccessNFCPage	The requested 256 byte NAND flash controller sub-page to be accessed
uwAccessNANDPage	The requested NAND flash page to be accessed.
uwAccessNANDBlock	The requested NAND flash block to be accessed.

ADI_BOOT_NAND_ADDRESS

```
typedef struct   {
   void *pSource;
   u32 udMainOffset;
   u32 udPrefetchOffset;
   u16 uwNumAddressCycles;
   u16 uwNumCommands;
   u16 uwSerialAccess;
   ADI_BOOT_NAND *pNandInfo
   #pragma align 4
   u8 ubCommand0;
   u8 ubAddress0;
   u8 ubAddress1;
```

```
    u8 ubAddress2;
    u8 ubAddress3;
    u8 ubAddress4;
    u8 ubCommand1;
}   ADI_BOOT_NAND_ADDRESS;
```

`ADI_BOOT_NAND_ADDRESS` is modified when the NAND flash boot kernel decodes the source address provided by the main kernel. When a booting feature is used that detects bad blocks or uncorrectable errors, offsets for addressing alternative blocks are applied. When the address is decoded, the structure is filled with the NAND flash controller commands and address cycles needed for retrieving the required data.

For supported small-page NAND flash devices, the number of address cycles is always four and the number of command cycles is one. For large-page NAND flash devices, the default number of address cycles is five. Since the upper addressing boundaries of the NAND flash device cannot be determined from the electronic signature, the kernel is unable to calculate the exact number of address cycles required to perform a read from the NAND flash. A majority of large-page NAND flash devices simply ignore any address cycles on a page read command that are not required. If a NAND flash device is not capable of ignoring the additional address cycles and it requires less than the default five address cycles for a page read operation then the device cannot be supported for NAND boot functionality. To remove the redundant address cycles, the required number of address cycles can be reconfigured within an initialization file executed before loading the main application.

Table 17-21. Structure Variables, `ADI_BOOT_NAND_ADDRESS`

Variable	Description
`pSource`	The source address to be accessed.
`udMainOffset`	The current block offset applied to data loaded into the main buffer.
`udPrefetchOffset`	The current block offset applied to data loaded into the prefetch buffer.

Table 17-21. Structure Variables, ADI_BOOT_NAND_ADDRESS (Continued)

Variable	Description
uwNumAddressCycles	The number of address cycles required to access the NAND flash device. This is set to 4 for small-page device booting and 5 for large-page devices.
uwNumCommands	The number of command cycles required to perform a read access from the NAND flash device. This parameter is set to 1 for small-page devices and 2 for large-page devices.
uwSerialAccess	Indicates that the next read access is from the next sequential 256 byte page to the previous access. This allows for the removal of the issuing of a read transaction thus optimizing throughput without waiting on unnecessary ready/$\overline{\text{busy}}$ assertions.
pNandInfo	Pointer to ADI_BOOT_NAND structure
ubCommand0	The first command to be issued to perform a page read from the NAND flash device.
ubAddress0	The first address cycle issued when performing a page read command.
ubAddress1	The second address cycle issued when performing a page read command.
ubAddress2	The third address cycle issued when performing a page read command.
ubAddress3	The fourth address cycle issued when performing a page read command.
ubAddress4	The fifth address cycle issued when performing a page read command.
ubCommand1	The second command to be issued to perform a page read from the NAND flash device. Only used for large-page devices.

ADI_BOOT_NAND_ECC

```
typedef struct   {
   #pragma align 4
   u16 uwIndex;
   u32 udNFCParity[32];
   u16 uwError;
   u16 uwBlockSkipFeature;
   u16 uwBlockModifier;
   u16 uwMaxCopies;
   u16 uwCurrentCopy;
}   ADI_BOOT_NAND_ECC;
```

This structure provides stack storage for the error correction parity data read from the spare area of a page when an access to a new NAND flash page is detected. The spare area contains parity data for each 256 byte block in a page. This allows for error correction and detection to be performed on every 256 byte load from the NAND flash. Enough storage space is provided to support devices up to and including a page size of 8K bytes. `ADI_BOOT_NAND_ECC` also contains the fields that need to be modified to enable the NAND flash boot options that skip bad blocks or boot from mirror images of the original boot stream located in other memory blocks.

Table 17-22. Structure Variables, `ADI_BOOT_NAND_ECC`

Variable	Description
`nIndex`	Index used to access the udNFCParityArray
`udNFCParity`	A 32 deep long word array providing storage for up to 32 256-byte NAND Flash Controller error correction parity data. The array provides support for page sizes up to and including 8 Kbytes.
`uwError`	Error that was generated within the error correction routine. 0 – No Error 1 – Error found in parity data 2 – Uncorrectable error
`uwBlockSkipFeature`	Specifies the NAND flash boot technique to be implemented. Defaults to 0 unless otherwise altered through an initialization sequence. 0 – Sequential booting from a single boot stream. No bad-block checking performed. 1 – Block Skip Method, allowing for a single boot stream loaded to the NAND flash to skip bad blocks. 2 – Mirror Image Mode, allowing for booting from multiple copies of the application in the event that an uncorrectable error or error in the ECC parity data is detected.
`uwError`	Indicates the error returned from the error correction routine if one occurred. 0 – No error or correctable error. 1 – Error in ECC parity data. 2 – Uncorrectable error.

Table 17-22. Structure Variables, ADI_BOOT_NAND_ECC (Continued)

Variable	Description
uwBlockModifier	The number of blocks to skip if a bad block is detected. If uwBlockSkipFeature is 0 this value is ignored. For an uwBlockSkipFeature value of 1 this parameter must be 1. For an uwBlockSkipFeature of 2 this parameter may be any value indicating the number of blocks between multiple copies of the application.
uwMaxCopies	The number of copies of the application stored in the NAND flash device. Only applicable if uwBlockSkipFeature is 2.
uwCurrentCopy	Indicates the current copy of the application that is being accessed. Only applicable if uwBlockSkipFeature is 2.

Callable ROM Functions for Booting

The following functions support boot management.

BFROM_FINALINIT

Entry address: 0xEF00 0002

Arguments: no arguments

C prototype: `void bfrom_FinalInit (void);`

The `bfrom_FinalInit` function never returns. It only executes a `JUMP` to the address stored in `EVT1`.

BFROM_PDMA

Entry address: 0xEF00 0004

Arguments: pointer to `ADI_BOOT_DATA` in `R0`

C prototype: `void bfrom_PDma (ADI_BOOT_DATA *p);`

This is the load function for peripherals such as SPI and UART that support DMA in their boot modes.

BFROM_MDMA

Entry address: 0xEF00 0006

Arguments: pointer to `ADI_BOOT_DATA` in `R0`

C prototype: `void bfrom_MDma (ADI_BOOT_DATA *p);`

This is the load function used for memory boot modes including the FIFO mode. This routine is also reused when the `BFLAG_FILL` or the `BFLAG_INDIRECT` flags are specified.

BFROM_MEMBOOT

Entry address: 0xEF00 0008

Arguments:

pointer to boot stream in `R0`

`dFlags` in `R1`

`dBlockCount` in `R2`

`pCallHook` passed over the stack in [FP+0x14]

updated block count returned in `R0`

C prototype:

```
s32 bfrom_MemBoot (void* pBootStream, s32 dFlags,
 s32 dBlockCount, ADI_BOOT_HOOK_FUNC* pCallHook);
```

This routine processes any boot stream that maps to the Blackfin memory starting from address `pBootStream`.

To boot a new application that may overwrite the calling application, the dFlags word is usually zero. When done, the routine does not return, but jumps to the EVT1 vector address. If the BFLAG_RETURN flag is set, an RTS is executed instead and the routine returns to the parent function. In this way, fractions of an application can be loaded.

If the dBlockCount parameter is zero or a positive value, all boot blocks are processed until the BFLAG_FINAL flag is detected. If dBlockCount is a negative value, the negative number represents the number of blocks to be booted. For example, –1 causes the kernel to return immediately, –2 processes only one block.

The routine returns the updated source address pSource of the boot stream (for example, the first unused address after the processed boot stream).

The BFLAG_NEXTDXE flag suppresses boot loading. The boot kernel steps through the boot stream by analyzing the next-DXE pointers (in the ARGUMENT field of a BFLAG_FIRST block) and jumping to the next DXE. Assuming that the boot image is a chained list of boot streams, the boot kernel returns the absolute start address of the requested boot stream. In this example, the start address of the third boot stream (DXE) in a flash device is returned.

```
bfrom_MemBoot((void*)0x20000000,
BFLAG_RETURN|BFLAG_NEXTDXE,-3, NULL);
```

In the above example, the routine would return 0x2000 0000 when dBlockCount was set to –1. If the parameter dBlockCount is zero or positive when used along with the BFLAG_NEXTDXE command, the kernel returns when the BFLAG_FIRST flag on a header in the next-DXE chain is not set.

If the BFLAG_HOOK switch is set, the memboot routine call (pCallHook routine) after the ADI_BOOT_DATA structure is filled with default values. It then can overrule the default settings of the structure.

BFROM_TWIBOOT

Entry address: 0xEF00 000C

Arguments:

TWI address in R0

dFlags in R1

dBlockCount in R2

pCallHook passed over the stack in [FP+0x14]

updated block count returned in R0

C prototype:

```
s32 bfrom_TwiBoot (s32 dTwiAddress, s32 dFlags,
 s32 dBlockCount, ADI_BOOT_HOOK_FUNC* pCallHook);
```

This routine processes boot streams residing in TWI memories. It differs from the BFROM_MEMBOOT routine in that some functionality is TWI specific.

Additional bits in the dFlags word are relevant. The user should always set the BFLAG_PERIPHERAL flag but never the BFLAG_SLAVE bit. The BFLAG_TYPE tells the boot kernel when addressing mode is required for the TWI memory. The boot kernel derives the values for the TWI0_CONTROL and TWI0_CLKDIV registers from the lower four bits of the dFlags word. See Chapter 23, “Two Wire Interface Controller”.

BFROM_SPIBOOT

Entry address: 0xEF00 000A

Arguments:

SPI address in R0

dFlags in R1

dBlockCount in R2

pCallHook passed over the stack in [FP+0x14]

updated block count returned in R0

C prototype:

```
s32 bfrom_SpiBoot (s32 dSpiAddress, s32 dFlags,
 s32 dBlockCount, ADI_BOOT_HOOK_FUNC* pCallHook);
```

This SPI master boot routine processes boot streams residing in SPI memories, using the SPI controller. It differs from the BFROM_TWIBOOT routine in that some functionality is SPI specific. The fourth argument pCallHook is passed over the stack. It provides a hook to call a callback routine after the ADI_BOOT_DATA structure is filled with default values. For example, the pCallHook routine may overwrite the default value of the uwSsel value in the ADI_BOOT_DATA structure. The coding follows the rules of uwHWAIT (see "Boot Host Wait (HWAIT) Feedback Strobe" on page 17-31). A value of 0x0501 represents GPIO PG1 ($\overline{\text{SPISEL1}}$), 0x060C represents PF12($\overline{\text{SPISEL2}}$) and so on.

Additional bits in the dFlags word are relevant. The user should always set the BFLAG_PERIPHERAL flag but never the BFLAG_SLAVE bit. The BFLAG_NOAUTO flag instructs the system to skip the SPI device detection routine. The BFLAG_TYPE then tells the boot kernel what addressing mode is required for the SPI memory. (see "SPI Device Detection Routine" on page 17-67). The BFLAG_FASTREAD flag controls whether standard SPI read (0x3 command) or fast read (0xB) is performed. The boot kernel writes the lower bits of the dFlags word to the SPI_BAUD registers.

BFROM_OTPBOOT

Entry address: 0xEF00 000E

Arguments:

OTP byte address in R0

dFlags in R1

dBlockCount in R2

pCallHook passed over the stack in [FP+0x14]

Updated block count returned in R0

C prototype:

```
s32 bfrom_OtpBoot (s32 dOtpAddress, s32 dFlags,
 s32 dBlockCount, ADI_BOOT_HOOK_FUNC* pCallHook);
```

This OTP boot routine processes boot streams residing in the on-chip, serial OTP memory. Unlike the bfrom_OtpRead() function which uses the half-page addressing method, this one requires byte addressing. For example, set the dOtpAddress argument to 0x400 to process a boot stream starting from OTP page 0x40. Remember that one OTP page spans 16 bytes.

BFROM_NANDBOOT

Entry address: 0xEF00 0010

Arguments:

NAND Flash address in R0

dFlags in R1

dBlockCount in R2

pCallHook passed over the stack in [FP+0x14]

updated block count returned in R0

C prototype:

```
s32 bfrom_NandBoot(s32 dNandAddress,
 s32 dFlags, s32 dBlockCount, ADI_BOOT_HOOK_FUNC *pCallHook)
```

This NAND flash boot routine processes boot streams residing in NAND flash memories, using the NAND Flash Controller. Some functionality is NAND flash specific.

Additional bits in the `dFlags` word are relevant. When the `BFLAG_NOAUTO` flag is set the `BFLAG_TYPE` field is used to indicate whether the connected NAND flash is a small-page or large-page device.

`BFLAG_TYPE` = 00 (`BFLAG_TYPE1`) indicates small-page NAND Flash

`BLAG_TYPE` = 01 (`BFLAG_TYPE2`) indicates large-page NAND Flash

`BFLAG_TYPE` — values of 11 and 10 are reserved

Detection of a reserved value results in a call to the error handler.

In the event the `NFC_CTL` register is set to the default reset value of 0x0200 prior to the call to `bfrom_NandBoot()`, the read and write delay strobes of the `NFC_CTL` register will each be set to 3 providing t_{RP} and t_{WP} timings of four `SCLK` cycles.

BFROM_BOOTKERNEL

Entry address: 0xEF00 0020

Arguments:

pointer to `ADI_BOOT_DATA` in `R0`

returns updated source address `pSource` in `R0`

C prototype:

```
s32 bfrom_BootKernel (ADI_BOOT_DATA *p);
```

This ROM entry provides access to the raw boot kernel routine. It is the user's responsibility to initialize the items passed in the ADI_BOOT_DATA structure. Pay particular attention that the function pointers (pLoadFunction, and pErrorFunction) point to functional routines.

BFROM_CRC32

Entry address: 0xEF00 0030

Arguments:

pointer to look-up table in R0

pointer to data in R1

dByteCount in R2

initial CRC value in R0

CRC value returned in R0

C prototype:

```
s32 bfrom_Crc32 (s32 *pLut, void *pData,
 s32 dByteCount, s32 dInitial);
```

This routine calculates the CRC32 checksum for a given array of bytes. The look-up table is typically generated by the BFROM_CRC32POLY routine. During the boot process this routine is called by the BFROM_CRC32CALLBACK routine. The dInitial value is normally set to zero unless the CRC32 routine is called in multiple slices. Then, the dInitial parameter expects the result of the former run.

BFROM_CRC32POLY

Entry address: 0xEF00 0032

Arguments:

pointer to look-up table in `R0`

polynomial in `R1`

updated block count returned in `R0`

C prototype:

```
s32 bfrom_Crc32Poly (unsigned s32 *pLut, s32 dPolynomial);
```

This function generates a 1024-byte look-up table from a given CRC polynomial. During the boot process this routine is hidden by the `BFROM_CRC32INITCODE` routine.

BFROM_CRC32CALLBACK

Entry address: 0xEF00 0034

Arguments:

pointer to `ADI_BOOT_DATA` in `R0`

pointer to `ADI_BOOT_BUFFER` in `R1* Callback Flags in R2`

C prototype:

```
s32 bfrom_Crc32Callback (ADI_BOOT_DATA *pBS, ADI_BOOT_BUFFER
*pCS, s32 dCbFlags);
```

This is a wrapper function that ensures the `BFROM_CRC32` subroutine fits into the boot process.

BFROM_CRC32INITCODE

Entry address: 0xEF00 0036

Arguments:
pointer to `ADI_BOOT_DATA` in `R0`

C prototype:

```
void bfrom_Crc32Initcode (ADI_BOOT_DATA *p);
```

This is an initcode residing in ROM with two jobs:

Register `BFROM_CRC32CALLBACK` as a callback routine to the `pCallback` pointer in `ADI_BOOT_DATA`.

Call `BFROM_CRC32POLY` to generate the look-up table.

This function is unlikely to be called by user code directly. This function is called as an initcode during the boot process when the CRC calculation is desired. See "CRC Checksum Calculation" on page 17-46 for details.

Programming Examples

System Reset

To perform a system reset, use the code shown in Listing 17-1 or Listing 17-2.

Listing 17-1. System Reset in Assembly

```
#include <blackfin.h>
P0.L = LO(BFROM_SYSCONTROL);
P0.H = HI(BFROM_SYSCONTROL);
R0.L = LO(SYSCTRL_SYSRESET);
R0.H = HI(SYSCTRL_SYSRESET);
R1 = 0;
R2 = 0;
CALL (P0);
```

Listing 17-2. System Reset in C Language

```
bfrom_SysControl(SYSCTRL_SYSRESET, 0, NULL);
```

Exiting Reset to User Mode

To exit reset while remaining in user mode, use the code shown in Listing 17-3.

Listing 17-3. Exiting Reset to User Mode

```
_reset:   P1.L = LO(_usercode); /* Point to start of user code */
P1.H = HI(_usercode);
RETI = P1;            /* Load address of _start into RETI */
RTI;                  /* Exit reset priority */
_reset.end:
_usercode:                /* Place user code here */
...
```

The reset handler most likely performs additional tasks not shown in the examples above. Stack pointers and `EVTx` registers are initialized here.

Exiting Reset to Supervisor Mode

To exit reset while remaining in supervisor mode, use the code shown in Listing 17-4.

Listing 17-4. Exiting Reset by Staying in Supervisor Mode

```
_reset:
   P0.L = LO(EVT15);  /* Point to IVG15 in Event Vector Table */
   P0.H = HI(EVT15);
   P1.L = LO(_isr_IVG15); /* Point to start of IVG15 code */
   P1.H = HI(_isr_IVG15);
   [P0] = P1;         /* Initialize interrupt vector EVT15 */
   P0.L = LO(IMASK);  /* read-modify-write IMASK register */
   R0 = [P0];         /* to enable IVG15 interrupts */
   R1 = EVT_IVG15 (Z);
```

```
    R0 = R0 | R1;       /* set IVG15 bit */
    [P0] = R0;          /* write back to IMASK */
    RAISE 15;           /* generate IVG15 interrupt request */
   /* IVG 15 is not served until reset handler returns */
    P0.L = LO(_usercode);
    P0.H = HI(_usercode);
RETI = P0;          /* RETI loaded with return address */
RTI;                /* Return from Reset Event */
_reset.end:
_usercode:              /* Wait in user mode till IVG15 */
   JUMP _usercode;      /* interrupt is serviced */
_isr_IVG15:             /* IVG15 vectors here due to EVT15 */
...
```

Initcode (SDRAM Controller Setup)

Listing 17-5 shows an example of initcode to setup the SDRAM controller. The SDRAM controller must be initialized before data can be booted into it. Therefore, the SDRAM controller is typically initialized by an initcode or by the preboot functionality. The following initcode example assumes that the preboot did not do the job.

Listing 17-5. Example Initcode (SDRAM Controller Setup) in C Language

```
#include <cdefBF527.h>
void init_SDRAM(void)
{
    while((*pEBIU_SDSTAT & SDCI) == 0){}
    /* clear SDRAM EAB sticky error status (W1C) */
    *pEBIU_SDSTAT |= SDEASE;
    /* SDRAM Refresh Rate Control Register */
    *pEBIU_SDRRC = 0x03F6;
    /* SDRAM Memory Bank Control Register */
```

```
    *pEBIU_SDBCTL = (EBE|EBSZ_64|EBCAW_10);
    /* SDRAM Memory Global Control Register */
    *pEBIU_SDGCTL =
(SCTLE|PSS|TWR_2|TRCD_3|TRP_3|TRAS_6|PASR_ALL|CL_3);
    /* Finalize SDC initialization */
    pTmp = (u16*) 0x0;
    *pTmp = 0xBEEF;
    while((*pEBIU_SDSTAT & SDRS) == 1){}
}
```

Listing 17-6. Example Initcode (SDRAM Controller Setup) in Assembly

```
#include <defBF527.h>
/* Load Immediate 32-bit value into data or address register */
#define IMM32(reg,val) reg##.H=hi(val);reg##.L=lo(val)
.SECTION L1_code;
init_SDRAM:
link 0;
[--SP] = ASTAT;
[--SP] = (R7:7, P5:4);
IMM32 (P5, EBIU_SDRRC);
PollSdcIdle:
R7 = w[P5 + EBIU_SDSTAT - EBIU_SDRRC] (z);
CC = bittst(R7,bitpos(SDCI));
if!CC jump PollSdcIdle;
/* clear SDRAM EAB sticky error status (W1C) */
R7 = SDEASE(z);
w[P5 + EBIU_SDSTAT - EBIU_SDRRC] = R7;
/* SDRAM Refresh Rate Control Register */
R7.L = 0x03F6;
w[P5 + EBIU_SDRRC - EBIU_SDRRC] = R7;
/* SDRAM Memory Bank Control Register */
w[P5 + EBIU_SDBCTL - EBIU_SDRRC] = R7;
R7.L = (EBE|EBSZ_64|EBCAW_10);
```

```
/* SDRAM Memory Global Control Register */
IMM32(R7,(SCTLE|PSS|TWR_2|TRCD_3|TRP_3|TRAS_6|PASR_ALL|CL_3));
[P5 + EBIU_SDGCTL - EBIU_SDRRC] = R7;
/* Finalize SDC initialization */
/* a transfer is required to finalize SDC initialization! */
IMM32(P4,0x4);
nop;
R7 = [P4];
PollSdcPowerUpFinished:
R7 = w[P5 + EBIU_SDSTAT - EBIU_SDRRC] (z);
CC = bittst(R7,bitpos(SDRS));
if CC jump PollSdcPowerUpFinished;
(R7:7,P5:4) = [SP++];
ASTAT = [SP++];
unlink;
rts;
init_SDRAM.end:
```

Since this initcode needs to be executed only once, it can be volatile and can be overwritten by other boot blocks.

Initcode (Power Management Control)

The following examples show how to change PLL and the voltage regulator within an initcode. The examples assume that the preboot did not do the job already.

Because the ADSP-BF522, ADSP-BF524, and ADSP-BF526 Blackfins are low power processors, the maximum clock (~80MHz) of the SDRAM controller is lower than the maximum possible system clock (133MHz). See the current data sheets for the actual values if SDRAM is in use.

The ADSP-BF522, ADSP-BF524, and ADSP-BF526 processors do not have an on-chip voltage regulator. Set the `bfrom_SysControl` option to `SYSCTRL_EXTVOLTAGE`.

Listing 17-7. Changing PLL and Voltage Regulator in C Language

```
#include <cdefBF527.h>
#include <bfrom.h>
void init_DPM(ADI_BOOT_DATA* pBS)
{
   ADI_SYSCTRL_VALUES init_DPM;
   init_DPM.uwVrCtl = (CLKBUFOE | VLEV_120 | FREQ);
   init_DPM.uwPllCtl = SET_MSEL(21);
   init_DPM.uwPllDiv = (SET_SSEL(4) | CSEL_DIV1);
   init_DPM.uwPllLockCnt = 0x0200;
   bfrom_SysControl(SYSCTRL_VRCTL | SYSCTRL_INTVOLTAGE |
SYSCTRL_PLLCTL | SYSCTRL_PLLDIV | SYSCTRL_LOCKCNT |
SYSCTRL_WRITE, &init_DPM, NULL);
}
```

Listing 17-8. Changing PLL and Voltage Regulator in Assembly

```
#include <cdefBF527.h>
#include <bfrom.h>
.import "bfrom.h";
/* Load Immediate 32-bit value into data or address register */
#define IMM32(reg,val) reg##.H=hi(val); reg##.L=lo(val)
.SECTION L1_code;
init_DPM:
link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:5);
SP += -12;
R0.L = (CLKBUFOE | VLEV_120 | FREQ);
```

```
w[FP+-sizeof(ADI_SYSCTRL_VALUES) + offset of
(ADI_SYSCTRL_VALUES,uwVrCtl)] = R0;
R0.L = SET_MSEL(21);
w[FP+-sizeof(ADI_SYSCTRL_VALUES) + offset of
(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;
R0.L = (SET_SSEL(4) | CSEL_DIV1);
w[FP+-sizeof(ADI_SYSCTRL_VALUES) + offset of
(ADI_SYSCTRL_VALUES,uwPllDiv)] = R0;
R0.L = 0x0200;
w[FP+-sizeof(ADI_SYSCTRL_VALUES) + offset of
(ADI_SYSCTRL_VALUES,uwPllLockCnt)] = R0;
R0 = (SYSCTRL_VRCTL | SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_PLLDIV | SYSCTRL_LOCKCNT | SYSCTRL_WRITE);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P5,BFROM_SYSCONTROL);
call(P5);
SP += 12;
(R7:0,P5:5) = [SP++];
unlink;
rts;
init_DPM.end:
```

Care must be taken that the reprogramming of the PLL does not break the communication with the booting host. For example, in the case of UART boot, the `UARTx_DLL` and `UARTx_DLH` registers must be updated to keep the old bit rate.

Initcode (NAND Flash Boot Mode Configuration)

Listing 17-9 shows an example of initcode to enable the advanced options available for NAND flash boot mode. The initcode is loaded while the NAND flash boot kernel is configured for the default boot mode. In this example, after the initcode sequence is executed, the NAND flash boot

kernel is in Multiple Image Mode. This example also alters the number of address cycles for further accesses, which further optimizes the boot kernel for the attached NAND flash device.

Listing 17-9. Initcode Options with NAND Flash Boot Mode

```
#include <bfrom.h>
void initcode(ADI_BOOT_DATA* pBS)
{
    /* Create a pointer to the ADI_BOOT_NAND structure */
    ADI_BOOT_NAND *pNS;
    /* Set the pointer to ADI_BOOT_NAND */
    pNS = pBS->dUserLong;
   /* NAND Boot Kernel Configuration
      Mode: Multiple Image Mode
      Number of blocks between each image: 10
      Number of images: 4
      Number of address cycles: 4 */
    pNS->EccParity.uwBlockSkipFeature = 2;
    pNS->EccParity.uwBlockModifier = 10;
    pNS->EccParity.uwMaxCopies = 3;
    pNS->AddressCycles.uwNumAddressCycles = 4;
}
```

Quickboot With Restore From SDRAM

This example could be part of an advanced power saving concept. Assume the Blackfin is waking up from hibernate and processing any master boot mode. If the SDRAM has not been shut down, but was put in self-refresh mode, the content of the SDRAM will still be valid after wake up. The boot process would only have to initialize on-chip memories. Several boot blocks might be tagged by the `BFLAG_QUICKBOOT` flag.

Some applications might use a power-down handler that saves the contents of L1 memory to SDRAM before entering the hibernate state. Listing 17-10 assumes a suitable power-down handler was present that generated a partial boot stream in SDRAM at address 0x0001 0000 containing all the instructions required to restore the L1 memory contents.

Listing 17-10. Quickboot with Restore from SDRAM

```
void L1_recovery_initcode (ADI_BOOT_DATA *pBS)
{
    if (pBS->dFlags & BFLAG_WAKEUP) {
    bfrom_MemBoot((void*)0x00010000, BFLAG_RETURN, NULL);
    }
}
```

The boot stream generated at 0x0001 0000 will only be processed upon a wake-up condition. The `BFLAG_RETURN` ensures that the new instance of the boot kernel returns to the initcode rather than jumps to the `EVT1` vector.

XOR Checksum

Listing 17-11 illustrates how an initcode can be used to register a callback routine. The routine is called after each boot block that has the `BFLAG_CALLBACK` flag set. The calculated XOR checksum is compared against the block header `ARGUMENT` field. When the checksum fails, this example goes into idle mode. Otherwise control is returned to the boot kernel.

Since this callback example accesses the data after it is loaded, it would fail if the target address were in L1 instruction space. Therefore the `BFLAG_INDIRECT` flag should also be set. The `xor_callback` routine could then perform the checksum calculation at an intermediate storage place. The boot kernel transfers the data from the temporary buffer to the final destination after the callback routine returns.

In general, the block size is bigger than the size of the temporary buffer. Therefore, the boot kernel may need to divide the processing of a single block into multiple steps. The callback routine may also need to be invoked multiple times—every time the temporary buffer is filled up and once for the remaining bytes. The boot kernel passes the `dFlags` parameter, so that the callback routines knows whether it is called the first time, the last time or neither. The `dUserLong` variable in the `ADI_BOOT_DATA` structure is used to store the intermediate results between function calls.

Listing 17-11. XOR Checksum

```
s32 xor_callback(ADI_BOOT_DATA* pBS, ADI_BOOT_BUFFER* pCS, s32
dFlags)
{
    s32 i;
    if ((pCS!= NULL) && (pBS->pHeader!= NULL)) {
        if (dFlags & CBFLAG_FIRST) {
        pBS->dUserLong = 0;
        }
        for (i=0; i<pCS->dByteCount/sizeof(s32); i++)
{
        pBS->dUserLong^= ((s32 *)pCS->pSource)[i];
        }
        if (dFlags & CBFLAG_FINAL) {
        if (pBS->dUserLong!= pBS->pHeader->dArgument) {
        idle ();
        }
        }
    }
    return 0;
}
void xor_initcode (ADI_BOOT_DATA *pBS)
{
```

```
    pBS->pCallBackFunction = xor_callback;
}
```

Note that the callback routine is not volatile. It should not be overwritten by subsequent boot blocks. It can, however, be overwritten after processing the last block with `BFLAG_CALLBACK` flag set.

The checksum algorithm must be booted first and cannot protect itself. Problems can be avoided by letting initcode and callback execute directly from off-chip flash memory. The ADSP-BF52x processors provide a CRC32 checksum algorithm in the on-chip L1 instruction ROM, that can be used for booting under this scenario. For more information see "CRC Checksum Calculation" on page 17-46.

Direct Code Execution

This code example illustrates how to instruct the VisualDSP++ tools to generate a flash image that causes the boot kernel to start code execution at flash address 0x2000 0020 rather than performing a regular boot. See "Direct Code Execution" on page 17-34.

First, a 32-byte data block is defined in an assembly file that contains the initial block.

```
.section bootblock;
.global _firstblock;
.var _firstblock[4] = 0xAD7BD006,
0x20000020, 0x00000010, 0x00000010;
```

Then, the linker is instructed to map the initial block to address 0x2000 0000 in the LDF file.

```
MEMORY
{
    MEM_ASYNC0
    {
        START(0x20000000)
        END(0x23FFFFFF)
```

```
        TYPE(ROM)
    WIDTH(8)
}
}
PROCESSOR p0
{
    RESOLVE(_firstblock,0x20000000)
    RESOLVE(start,0x20000020)
    KEEP(start,_firstblock)
SECTIONS
{
    flash
    {
        INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program))
        INPUT_SECTIONS($OBJECTS(bootblock))
} >MEM_ASYNC0
}
}
```

To invoke the elfloader utility, activate the meminit feature and use the command-line switches `-romsplitter` and `-maskaddr`. Refer to the application note *Running Programs from Flash on ADSP-BF533 Blackfin Processors (EE-239)* for further details.

Managing PBS Pages in OTP Memory

The following code snips illustrate how to read and write OTP memory, as it is required for the Preboot Settings (PBS). For detailed description of OTP API functions `bfrom_OtpCommand()`, `bfrom_OtpRead()` and `bfrom_OtpWrite()` used here, see Chapter 4, “One-Time Programmable Memory”.

The first example reads PBS settings from OTP and stores them into an instance of the `ADI_PBS_BLOCK` structure. This is a union composite of the

ADI_PBS_HALFPAGES or the ADI_PBS_BITFIELDS types. These structure types are defined in the bfrom.h header file. The dPbsSet variable describes the set of PBS pages of interest. A 0x00 value reads from OTP pages 0x18 to 0x1B. A 0x01 value reads from OTP pages 0x1C to 0x1F and so on.

Listing 17-12. Reading a Set of PBS Pages from OTP Memory

```
#include <blackfin.h>
#include <bfrom.h>
ADI_PBS_BLOCK PBS;
u32 dPbsSet = 0;
bfrom_OtpCommand(OTP_INIT, OTP_INIT_VALUE);
bfrom_OtpRead(PBS00+dPbsSet*4,OTP_LOWER_HALF,
&(PBS.HalfPages.uqPbs00L));
bfrom_OtpRead(PBS00+dPbsSet*4,
OTP_UPPER_HALF,&(PBS.HalfPages.uqPbs00H));
bfrom_OtpRead(PBS01+dPbsSet*4,OTP_LOWER_HALF,
&(PBS.HalfPages.uqPbs01L));
bfrom_OtpRead(PBS01+dPbsSet*4,OTP_UPPER_HALF,
&(PBS.HalfPages.uqPbs01H));
bfrom_OtpRead(PBS02+dPbsSet*4,OTP_LOWER_HALF,
&(PBS.HalfPages.uqPbs02L));
bfrom_OtpCommand(OTP_CLOSE, 0);
```

The next example shows how PBS pages can be written.

Listing 17-13. Programming a Set of PBS Pages from OTP Memory

```
#include <blackfin.h>
#include <bfrom.h>
ADI_PBS_BLOCK PBS;
u32 dPbsSet = 0;
/* fill PBS with meaningful data */
```

```
bfrom_OtpCommand(OTP_INIT, OTP_INIT_VALUE);
bfrom_OtpWrite(PBS00+dPbsSet*4, OTP_LOWER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs00L));
bfrom_OtpWrite(PBS00+dPbsSet*4, OTP_UPPER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs00H));
bfrom_OtpWrite(PBS01+dPbsSet*4, OTP_LOWER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs01L));
bfrom_OtpWrite(PBS01+dPbsSet*4, OTP_UPPER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs01H));
bfrom_OtpWrite(PBS02+dPbsSet*4, OTP_LOWER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs02L));
bfrom_OtpWrite(PBS02+dPbsSet*4, OTP_UPPER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs02H));
bfrom_OtpWrite(PBS03+dPbsSet*4, OTP_LOWER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs03L));
bfrom_OtpWrite(PBS03+dPbsSet*4, OTP_UPPER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs03H));
bfrom_OtpCommand(OTP_CLOSE, 0);
```

If a set of PBS pages has been written earlier, but need to be replaced by a new set, the old PBS pages have to be invalidated. Do not use the `OTP_CHECK_FOR_PREV_WRITE` option in this case.

Listing 17-14. Invalidating a Set of PBS Pages

```
#include <blackfin.h>
#include <bfrom_h>
u32 dPbsSet = 0;
u64 dlInvalidate = (u64)0xC000000000000000;
bfrom_OtpWrite(PBS00+dPbsSet*4,
bfrom_OtpCommand(OTP_INIT, OTP_INIT_VALUE);
OTP_LOWER_HALF | OTP_NO_ECC, &dlInvalidate);
bfrom_OtpCommand(OTP_CLOSE, 0);
dPbsSet++;
/* write next set as in Listing x-2 */
```

For production you may want to lock the PBS pages to protect them from being overwritten in the field. This can be performed by the following instructions:

Listing 17-15. Write-protecting a Set of PBS Pages

```
#include <blackfin.h>
#include <bfrom.h>
u32 dPbsSet = 0;
bfrom_OtpCommand(OTP_INIT, OTP_INIT_VALUE);
bfrom_OtpWrite(PBS00+dPbsSet*4, OTP_LOCK, NULL);
bfrom_OtpWrite(PBS01+dPbsSet*4, OTP_LOCK, NULL);
bfrom_OtpWrite(PBS02+dPbsSet*4, OTP_LOCK, NULL);
bfrom_OtpWrite(PBS03+dPbsSet*4, OTP_LOCK, NULL);
bfrom_OtpCommand(OTP_CLOSE, 0);
```

When locking PBS pages remember to duplicate the active set of PBS pages best reliability. In the above examples, if the `dPbsSet*4` contains the final configuration, then program set 5 with the same data. For completeness, note that the above code example does not lock the ECC fields corresponding to the PBS pages. See Chapter 4, “One-Time Programmable Memory” for details.

18 DYNAMIC POWER MANAGEMENT

This chapter describes the dynamic power management functionality of the Blackfin processor and includes the following sections:

- "Phase Locked Loop and Clock Control" on page 18-1
- "Dynamic Power Management Controller" on page 18-7
  - "Operating Modes" on page 18-8
  - "Dynamic Supply Voltage Control" on page 18-16
- "System Control ROM Function" on page 18-30
- "PLL and VR Registers" on page 18-25
- "Programming Examples" on page 18-36

Phase Locked Loop and Clock Control

The input clock into the processor, CLKIN, provides the necessary clock frequency, duty cycle, and stability to allow accurate internal clock multiplication by means of an on-chip PLL module. During normal operation, the user programs the PLL with a multiplication factor for CLKIN. The resulting, multiplied signal is the voltage controlled oscillator (VCO) clock. A user-programmable value then divides the VCO clock signal to generate the core clock (CCLK).

A user-programmable value divides the VCO signal to generate the system clock (SCLK). The SCLK signal clocks the Peripheral Access Bus (PAB),

DMA Access Bus (DAB), External Access Bus (EAB), and the external bus interface unit (EBIU).

These buses run at the PLL frequency divided by 1–15 (SCLK domain). Using the SSEL parameter of the PLL divide register, select a divider value that allows these buses to run at or below the maximum SCLK rate specified in the processor data sheet.

To optimize performance and power dissipation, the processor allows the core and system clock frequencies to be changed dynamically in a "coarse adjustment." For a "fine adjustment," the PLL clock frequency can also be varied.

PLL Overview

To provide the clock generation for the core and system, the processor uses an analog PLL with programmable state machine control.

The PLL design serves a wide range of applications. It emphasizes embedded and portable applications and low cost, general-purpose processors, in which performance, flexibility, and control of power dissipation are key features. This broad range of applications requires a wide range of frequencies for the clock generation circuitry. The input clock may be a crystal, a crystal oscillator, or a buffered, shaped clock derived from an external system clock oscillator.

The PLL interacts with the Dynamic Power Management Controller (DPMC) block to provide power management functions for the processor. For information about the DPMC, see "Dynamic Power Management Controller" on page 18-7.

Subject to the maximum VCO frequency specified in the processor data sheet, the PLL supports a wide range of multiplier ratios and achieves multiplication of the input clock, CLKIN. To achieve this wide multiplication range, the processor uses a combination of programmable dividers in the PLL feedback circuit and output configuration blocks.

Figure 18-1 illustrates a conceptual model of the PLL circuitry, configuration inputs, and resulting outputs. In the figure, the VCO is an intermediate clock from which the core clock (CCLK) and system clock (SCLK) are derived.

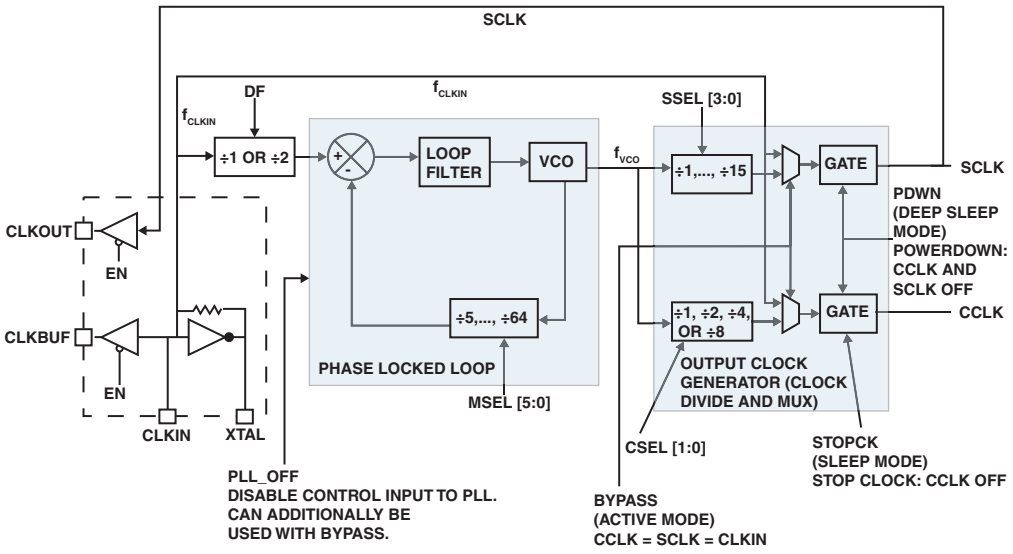


Figure 18-1. PLL Block Diagram

PLL Clock Multiplier Ratios

The PLL control register (PLL_CTL) governs the operation of the PLL. For details about the PLL_CTL register, see "PLL_CTL Register" on page 18-26.

The divide frequency (DF) bit and multiplier select (MSEL[5:0]) field configure the various PLL clock dividers:

- DF enables the input divider
- MSEL[5:0] controls the feedback dividers

The reset value of MSEL is 0x5 for ADSP-BF523/5/7 and 0x6 for ADSP-BFBF522/4/6. This value can be reprogrammed at startup in the boot code.

Table 18-1 illustrates the VCO multiplication factors for the various MSEL and DF settings.

As shown in the table, different combinations of MSEL[5:0] and DF can generate the same VCO frequencies. For a given application, one combination may provide lower power or satisfy the VCO maximum frequency. Under normal conditions, setting DF to 1 typically results in lower power dissipation. See the processor data sheet for maximum and minimum frequencies for CLKIN, CCLK, and VCO.

Table 18-1. MSEL Encodings

Signal name	VCO Frequency	
MSEL[5:0]	DF = 0	DF = 1
5	5x	2.5x
6	6x	3x
N = 7–62	Nx	0.5Nx
63	63x	31.5x
0	64x	32x

The PLL control (PLL_CTL) register controls operation of the PLL (see Figure 18-5 on page 18-26). Note that changes to the PLL_CTL register do not take effect immediately. In general, the PLL_CTL register is first programmed with a new value, and then a specific PLL programming sequence must be executed to implement the changes. This is handled automatically by the system control ROM function (bfrom_SysControl()) as described in "System Control ROM Function" on page 18-30.

Core Clock/System Clock Ratio Control

Table 18-2 describes the programmable relationship between the VCO frequency and the core clock. Table 18-3 shows the relationship of the VCO frequency to the system clock. Note the divider ratio must be chosen to limit the SCLK to a frequency specified in the processor data sheet. The SCLK drives all synchronous, system-level logic.

The divider ratio control bits, CSEL and SSEL, are in the PLL divide (PLL_DIV) register. For information about this register, see "PLL_DIV Register" on page 18-26.

The reset value of CSEL[1:0] is 0x0, and the reset value of SSEL[3:0] is 0x4. These values can be reprogrammed at startup by the boot code.

By updating PLL_DIV with an appropriate value, you can change the CSEL and SSEL value dynamically. Note the divider ratio of the core clock can never be greater than the divider ratio of the system clock. If the PLL_DIV register is programmed to illegal values, the SCLK divider is automatically increased to be greater than or equal to the core clock divider.

Unlike writing the PLL_CTL register, the PLL_DIV register can be programmed at any time to change the CCLK and SCLK divide values without entering the PLL programing sequence.

Table 18-2. Core Clock Ratio

Signal Name CSEL[1:0]	Divider Ratio VCO/CCLK	Example Frequency Ratios (MHz) VCO	CCLK
00	1	300	300
01	2	600	300
10	4	600	150
11	8	400	50

As long as the MSEL and DF control bits in the PLL control (PLL_CTL) register remain constant, the PLL is locked.

Table 18-3. System Clock Ratio

Signal Name SSEL[3:0]	Divider Ratio VCO/SCLK	Example Frequency Ratios (MHz) VCO	SCLK
0000	Reserved	N/A	N/A
0001	1:1	100	100
0010	2:1	200	100
0011	3:1	400	133
0100	4:1	500	125
0101	5:1	600	120
0110	6:1	600	100
N = 7–15	N:1	600	600/N

If changing the clock ratio via writing a new SSEL value into PLL_DIV, take care that the enabled peripherals do not suffer data loss due to SCLK frequency changes.

When changing clock frequencies in the PLL, the PLL requires time to stabilize and lock to the new frequency. The PLL lock count (PLL_LOCKCNT) register defines the number of CLKIN cycles that occur before the processor sets the PLL_LOCKED bit in the PLL_STAT register. When executing the PLL programming sequence, the internal PLL lock counter begins incrementing upon execution of the IDLE instruction. The lock counter increments by 1 each CLKIN cycle. When the lock counter has incremented to the value defined in the PLL_LOCKCNT register, the PLL_LOCKED bit is set.

See the processor data sheet for more information about PLL stabilization time and programmed values for this register. For more information about operating modes, see “Operating Modes” on page 18-8.

Dynamic Power Management Controller

The Dynamic Power Management Controller (DPMC) works in conjunction with the PLL, allowing the user to control the processor's performance characteristics and power dissipation dynamically. The DPMC provides these features that allow the user to control performance and power:

- Multiple operating modes – The processor works in four operating modes, each with different performance characteristics and power dissipation profiles. See "Operating Modes" on page 18-8.
- Peripheral clocks – Clocks to each peripheral are disabled automatically when the peripheral is disabled.
- Voltage control – The processor provides the option of external or internal regulator (ADSP-BF523/ADSP-BF525/ADSP-BF527 only) power supply. If the internal regulator is chosen, the on-chip switching regulator controller, with some external components, can generate internal voltage levels from the external supply.

  Depending on the needs of the system, the voltage level can be reduced to save power. See "Controlling the Internal Voltage Regulator" on page 18-19.

Operating Modes

The processor works in four operating modes, each with unique performance and power saving benefits. Table 18-4 summarizes the operational characteristics of each mode.

Table 18-4. Operational Characteristics

Operating Mode	Power Savings	PLL Status	PLL Bypassed	CCLK	SCLK	Allowed DMA Access
Full On	None	Enabled	No	Enabled	Enabled	L1
Active	Medium	Enabled [1]	Yes	Enabled	Enabled	L1
Sleep	High	Enabled	No	Disabled	Enabled	–
Deep Sleep	Maximum	Disabled	–	Disabled	Disabled	–

1 PLL can also be disabled in this mode.

Dynamic Power Management Controller States

Power management states are synonymous with the PLL control state. The active and full-on states of the DPMC/PLL can be determined by reading the PLL status register (see "PLL_STAT Register" on page 18-28). In these modes, the core can either execute instructions or be in the `IDLE` core state. If the core is in the `IDLE` state, it can be awakened by several sources (See Chapter 5, "System Interrupts" for details).

The following sections describe the DPMC/PLL states in more detail, as they relate to the power management controller functions.

Full-On Mode

Full-on mode is the maximum performance mode. In this mode, the PLL is enabled and not bypassed. Full-on mode is the normal execution state of the processor, with the processor and all enabled peripherals running at

full speed. The system clock (SCLK) frequency is determined by the SSEL-specified ratio to VCO. DMA access is available to L1 and external memories. From full-on mode, the processor can transition directly to active, sleep, or deep sleep modes, as shown in Figure 18-2 on page 18-12.

Active Mode

In active mode, the PLL is enabled but bypassed. Because the PLL is bypassed, the processor's core clock (CCLK) and system clock (SCLK) run at the input clock (CLKIN) frequency. DMA access is available to appropriately configured L1 and external memories.

In active mode, it is possible not only to bypass, but also to disable the PLL. If disabled, the PLL must be re-enabled before transitioning to full-on or sleep modes.

From active mode, the processor can transition directly to full-on, sleep, or deep sleep modes.

In this mode or in the transition phase to other modes, changes to MSEL are not latched by the PLL.

Sleep Mode

Sleep mode significantly reduces power dissipation by idling the processor core. The CCLK is disabled in this mode; however, SCLK continues to run at the speed configured by MSEL and SSEL bit settings. Since CCLK is disabled, DMA access is available only to external memory in sleep mode. From sleep mode, a wakeup event causes the processor to transition to one of these modes:

- Active mode if the BYPASS bit in the PLL_CTL register is set
- Full-on mode if the BYPASS bit is cleared

When sleep mode is exited, the processor resumes execution from the program counter value present immediately prior to entering sleep mode. Upon exiting sleep mode the user should clear the STOPCK bit.

Deep Sleep Mode

Deep sleep mode maximizes power savings by disabling the PLL, CCLK, and SCLK. In this mode, the processor core and all peripherals except the real-time clock (RTC) are disabled. DMA is not supported in this mode.

Deep sleep mode can be exited only by a hardware reset event or an RTC interrupt. A hardware reset begins the hardware reset sequence. For more information about hardware reset, see Chapter 5, "System Interrupts". An RTC interrupt causes the processor to transition to active mode, and execution resumes from where the program counter was when deep sleep mode was entered. If an interrupt is also enabled in SIC_IMASK, the vector is taken immediately after exiting deep sleep and the ISR is executed.

Note an RTC interrupt in deep sleep mode automatically resets some fields of the PLL control (PLL_CTL) register. See Table 18-5.

When in deep sleep mode, clocking to the SDRAM is turned off. Before entering deep sleep mode, software should ensure that important information in SDRAM is saved to a non-volatile memory and/or the SDRAM is placed into self-refresh mode.

Table 18-5. PLL_CTL Values after RTC Wakeup Interrupt

Field	Value
PLL_OFF	0
STOPCK	0
PDWN	0
BYPASS	1

Hibernate State

For lowest possible power dissipation, this state allows the internal supply (V_{DDINT}) to be powered down, while keeping the I/O supply (V_{DDEXT} and V_{DDMEM}) running. Although not strictly an operating mode like the four modes detailed above, it is illustrative to view it as such in the diagram of Figure 18-2 on page 18-12. Since this feature is coupled to the on-chip switching regulator controller (ADSP-BF523/ADSP-BF525/ADSP-BF527 only), it is discussed in detail in "Powering Down the Core (Hibernate State)" on page 18-22.

Operating Mode Transitions

Figure 18-2 on page 18-12 graphically illustrates the operating modes and transitions. In the diagram, ellipses represent operating modes and rectangles represent processor states. Arrows show the allowed transitions into and out of each mode or state.

For mode transitions, the text next to each transition arrow shows the fields in the PLL control (`PLL_CTL`) register that must be changed for the transition to occur. For example, the transition from full-on mode to sleep mode indicates that the `STOPCK` bit must be set to 1 and the `PDWN` bit must be set to 0.

For transitions to processor states, the text next to each transition arrow shows either a processor event (RTC wake up or hardware reset) or the fields in the voltage regulator control register (`VR_CTL`) that must be changed for the transition to occur.

For information about how to effect mode transitions, see "Programming Operating Mode Transitions" on page 18-14.

In addition to the mode transitions shown in Figure 18-2 on page 18-12, the PLL can be modified while in active operating mode. Changes to the PLL do not take effect immediately. As with operating mode transitions, the PLL programming sequence must be executed for these changes to

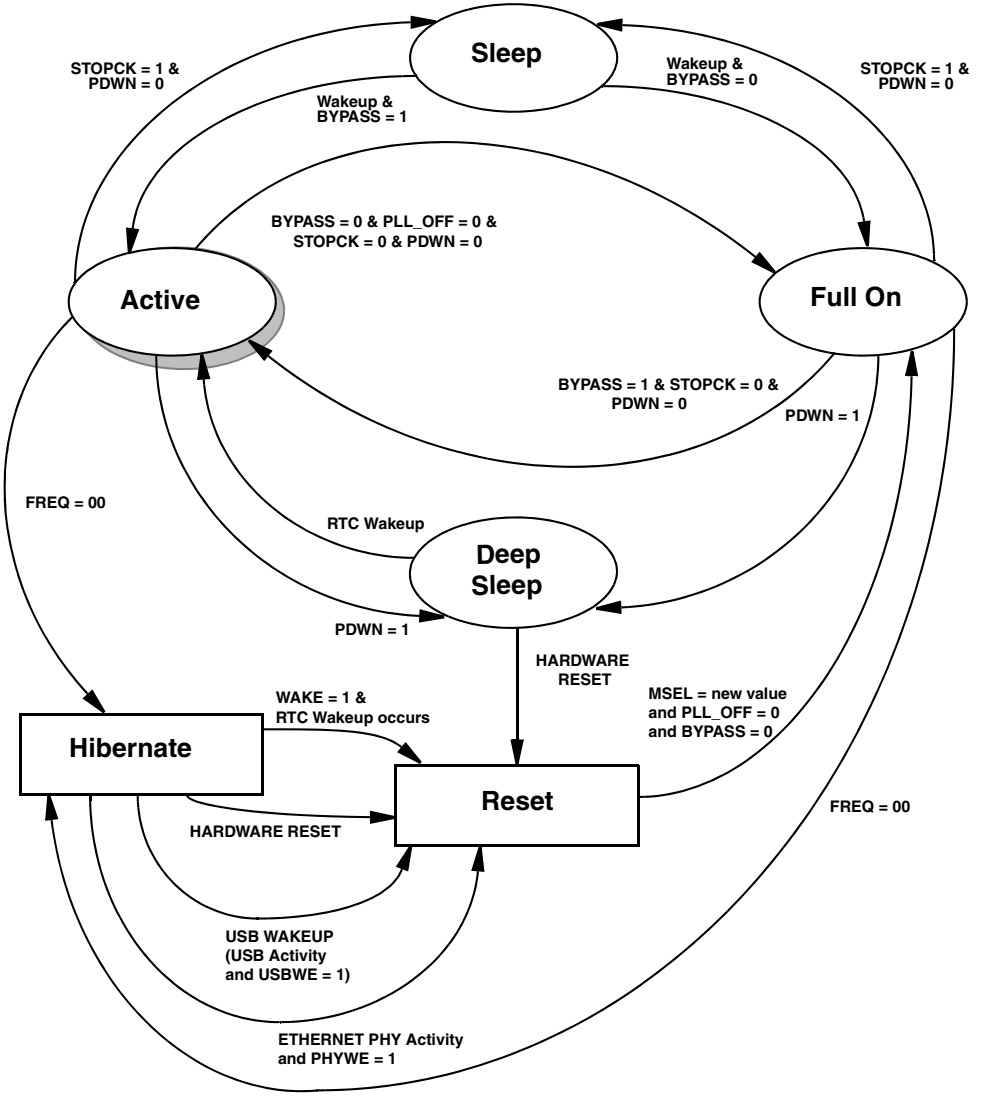


Figure 18-2. Operating Mode Transitions

take effect (see "Programming Operating Mode Transitions" on page 18-14).

- PLL disabled: In addition to being bypassed in the active mode, the PLL can be disabled.

  When the PLL is disabled, additional power savings are achieved although they are relatively small. To disable the PLL, set the PLL_OFF bit in the PLL_CTL register, and then execute the PLL programming sequence.

- PLL enabled: When the PLL is disabled, it can be re-enabled later when additional performance is required.

  The PLL must be re-enabled before transitioning to the full-on or sleep operating modes. To re-enable the PLL, clear the PLL_OFF bit in the PLL_CTL register, and then execute the PLL programming sequence.

- New multiplier ratio: The multiplier ratio can also be changed while in full-on mode.

  The PLL state automatically transitions to active mode while the PLL is locking. After locking, the PLL returns to full-on mode. To program a new CLKIN to VCO multiplier, write the new MSEL[5:0] and/or DF values to the PLL_CTL register; then execute the PLL programming sequence (see on page 18-14).

Table 18-6 summarizes the allowed operating mode transitions.

Attempting to cause mode transitions other than those shown in Table 18-6 causes unpredictable behavior.

Table 18-6. Allowed Operating Mode Transitions

New Mode	Current Mode			
	Full-On	**Active**	**Sleep**	**Deep Sleep**
Full On	–	Allowed	Allowed	Allowed
Active	Allowed	–	Allowed	Allowed
Sleep	Allowed	Allowed	–	–
Deep Sleep	Allowed	Allowed	–	–

Programming Operating Mode Transitions

The operating mode is defined by the state of the PLL_OFF, BYPASS, STOPCK, and PDWN bits of the PLL control (PLL_CTL) register. Merely modifying the bits of the PLL_CTL register does not change the operating mode or behavior of the PLL. Changes to the PLL_CTL register are realized only after a specific code sequence is executed. This sequence is managed by a user-callable routine in the on-chip ROM called bfrom_SysControl(). When calling this function, no further precautions have to be taken. See "System Control ROM Function" on page 18-30 for more information.

Failure to utilize the bfrom_SysControl() ROM function to modify PLL settings may result in processor malfunction.

If the PLL_CTL register changes include a new CLKIN to VCO multiplier or power is reapplied to the PLL, the PLL needs to relock. To relock, the PLL lock counter is cleared first, then starts incrementing once per CLKIN cycle. After the PLL lock counter reaches the value programmed in the PLL lock count (PLL_LOCKCNT) register, the PLL sets the PLL_LOCKED bit in

the PLL status (PLL_STAT) register, and the PLL asserts the PLL wake-up interrupt.

When the bfrom_SysControl() routine reprograms the PLL_CTL register with a new value, the bfrom_SysControl() routine executes a subsequent IDLE instruction and prevents all other system interrupt sources, other than the DPMC, from waking up the core from the IDLE state. If the lock counter expires, the PLL issues an interrupt, and the code execution continues the instruction after the IDLE instruction. Therefore, the system is in the new state by the time the bfrom_SysControl() routine returns.

If the new value written to the PLL_CTL or VR_CTL register is the same as the previous value, the PLL wake-up occurs immediately (PLL is already locked), but the core and system clock are bypassed for the PLL_LOCKCNT duration. For this interval, code executes at the CLKIN rate instead of the expected CCLK rate. Software guards against this condition by comparing the current value to the new value before writing the new value.

- When the wake-up signal is asserted, the code execution continues the instruction after the IDLE instruction, causing a transition to:
    - Active mode if BYPASS in the PLL_CTL register is set
    - Full-on mode if the BYPASS bit is cleared
- If the PLL_CTL register is programmed to enter the sleep operating mode, the processor transitions immediately to sleep mode and waits for a wake-up signal before continuing code execution. If the

PLL_CTL register is programmed to enter the deep sleep operating mode, the processor immediately transitions to deep sleep mode and waits for an RTC interrupt or hardware reset signal:

- An RTC interrupt causes the processor to enter active operating mode and to return from the bfrom_SysControl() routine.
- A hardware reset causes the processor to execute the reset sequence. For more information, see "Hardware Reset" on page 17-6.

If no operating mode transition is programmed, the PLL generates a wake-up signal, and the bfrom_SysControl() routine returns.

Dynamic Supply Voltage Control

In addition to clock frequency control, the processor provides the capability to run the core at different voltage levels. As power dissipation is proportional to the voltage squared, significant power reductions can be accomplished when lower voltages are used.

The processor uses multiple power domains. Each power domain has a separate V_{DD} supply. Note that the internal logic of the processor and much of the processor I/O can be run over a range of voltages. See the product data sheet for details on the allowed voltage ranges for each power domain and power dissipation data.

ADSP-BF523/ADSP-BF525/ADSP-BF527 feature an internal voltage regulator while ADSP-BF522/ADSP-BF524/ADSP-BF526 do not. Therefore, all references to the internal voltage regulator only apply to ADSP-BF523/ADSP-BF525/ADSP-BF527 processors.

Power Supply Management

The VRSEL pin selects between the external and the on-chip voltage regulator for the internal core power supply (ADSP-BF523/ ADSP-BF525/ ADSP-BF527 only). If the internal regulator is chosen (VRSEL = 0), the on-chip switching regulator controller, with some external components, can generate internal voltage levels from the external V_{DD} (V_{DDEXT}) supply as shown in Figure 18-3 on page 18-18 and Table 18-7 on page 18-20. This voltage level can be reduced to save power, depending upon the needs of the system.

In the internal voltage regulator mode, the SS/$\overline{\text{PG}}$ pin has soft start functionality. An external capacitor connected to this pin controls the in-rush current at start up. A larger capacitor allows for slower and smoother V_{DDINT} transitions, with smaller in-rush current. As a general indicator, a 500 pF capacitor results in approximately a 50 µS start-up time (V_{DDINT}

rising from 0 V to its programmed voltage), and the characteristic is fairly linear (such that doubling the capacitance doubles the start-up time).

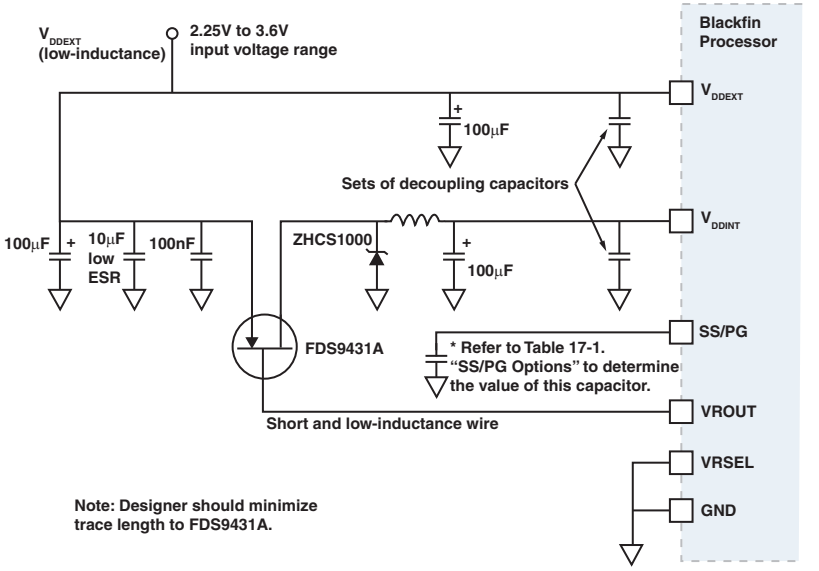


Figure 18-3. Processor Voltage Regulator

When increasing the V_{DDINT} voltage, the external FET switches on for a longer period. The V_{DDEXT} supply should have appropriate capacitive bypassing to enable it to provide sufficient current without drooping the supply voltage.

In external regulator mode (VRSEL = 1), the V_{DDINT} is supplied by an external regulator and the pin SS/$\overline{PG}$ is used to accept an active-low power-good indicator from the regulator. Note that the external regulator must comply with the V_{DDINT} specifications defined in the processor data sheet.

Controlling the Internal Voltage Regulator

The on-chip core voltage regulator controller manages the internal logic voltage levels for the V_{DDINT} supply. The voltage regulator control register (VR_CTL) controls the regulator (see Figure 18-9 on page 18-29). The state of the VR_CTL register is maintained during deep sleep modes and hibernate. It is only set to its reset value by a power up reset sequence. The VR_CTL register should not be written directly. Rather, the bfrom_SysControl() routine, which resides in the on-chip ROM, should be used to access the register.

Failure to utilize the bfrom_SysControl() ROM function to modify regulator settings may result in processor malfunction.

Changing Voltage on ADSP-BF523/ADSP-BF525/ADSP-BF527

Minor changes in operating voltage can be accommodated without requiring special consideration or action by the application program. See the processor data sheet for more information about voltage tolerances and allowed rates of change.

Reducing the processor's operating voltage to greatly conserve power or raising the operating voltage to greatly increase performance will probably require significant changes to the operating voltage level. To ensure predictable behavior the recommended procedure is to bring the processor to the sleep operating mode before substantially varying the voltage.

The recommended procedure is to follow the PLL programming sequence when varying the voltage. The four-bit voltage level (VLEV) field identifies

the nominal internal voltage level. Please refer to the processor data sheet for the applicable VLEV voltage range and associated voltage tolerances.

The VLEV should not be programmed to a value more than the maximum or less than the minimum value specified in the data sheet.

Table 18-7 lists the voltage level values for VLEV[3:0].

Table 18-7. VLEV Encodings

VLEV	Voltage
b#0000–b#0110	Reserved
b#0111	1.00 volts
b#1000	1.05 volts
b#1001 (ADSP-BF523/ADSP-BF525/ADSP-BF527 Default)	1.10 volts
b#1010	1.15 volts
b#1011	1.20 volts
b#1100	1.25 volts
b#1101	1.30 volts
b#1110	1.35 volts
b#1111	1.40 volts

After changing the voltage level in the VR_CTL register, the PLL automatically enters the active mode when the processor enters the IDLE state. At that point the voltage level changes and the PLL relocks with the new volt-

age. After the PLL_LOCKCNT has expired, the part returns to the full-on state.

After the voltage has been changed to the new level, the processor can safely return to any operational mode so long as the operating parameters, such as core clock frequency (CCLK), are within the limits specified in the processor data sheet for the new operating voltage level. The VSTAT bit in the PLL_STAT register can be used to indicate whether the V_{DDINT} is stable and ready to use.

When the internal voltage regulator is bypassed or not present and the V_{DDINT} voltage is applied by an external regulator—the bfrom_SysControl() routine must be called at startup or whenever the voltage changes at run time. Afterwards, the SYSCTRL_EXTVOLTAGE bit should be set along with the VLEV value corresponding to the externally applied voltage.

Changing Voltage on ADSP-BF522/ADSP-BF524/ADSP-BF526

When changing the voltage using an external regulator, a specific programming sequence must be followed.

Unlike other Blackfin derivatives that feature an internal voltage regulator; the voltage level for the ADSP-BF522/ADSP-BF524/ADSP-BF526 cannot be changed by programming the VR_CTL register. With an internal voltage regulator, the PLL would automatically enter the active mode when the processor enters the IDLE state. At that point the voltage level would change and the PLL would re-lock to the new voltage. After the PLL_LOCKCNT has expired, the part returns to the full-on state.

With an external voltage regulator, this sequence must be reproduced in the program code by the user. The PLL_LOCKCNT register cannot be used in this case, but the value is still needed for calculating the required delay. A larger PLL_LOCKCNT value may be necessary for changing voltages than

when changing just the PLL frequency. See the processor data sheet for details.

The processor must enter active mode before the user can access the external voltage regulator and program a new voltage level. See the data sheet of external voltage regulator for information on changing voltage levels. See the processor data sheet for more information about voltage tolerances and allowed rates of change.

The user must ensure a stable voltage and give the PLL time to re-lock at the new voltage level. This can be done by running the core in a loop for a certain amount of time before leaving active mode.

After the voltage has been changed to the new level, the processor can safely return to any operational mode—so long as the operating parameters, such as core clock frequency (CCLK), are within the limits specified in the processor data sheet for the new operating voltage level.

Please see "Changing Voltage Levels" on page 18-45 for more details on mode transitions and changing voltage levels.

The VSTAT bit in the PLL_STAT register can be used to indicate whether V_{DDINT} is stable and ready to use. The VSTAT bit works in conjunction with the $\overline{PG}$ (Power Good) input signal of the ADSP-BF522/ ADSP-BF524/ ADSP-BF526. The inverted version of a "power good" signal from the external regulator is fed to the ADSP-BF522/ ADSP-BF524/ADSP-BF526 to indicate that the voltage has reached its programmed value. That in turn will set the VSTAT bit, which should be considered the end of your "wait" state for the voltage regulator to settle.

Powering Down the Core (Hibernate State)

The internal supply regulator for the processor can be shut off by writing b#00 to the FREQ bits of the VR_CTL register, which disables CCLK and SCLK. Furthermore, it sets the internal power supply voltage (V_{DDINT}) to 0 V, eliminating any leakage currents from the processor. However, if CLKBUFOE

is set, the crystal oscillator and CLKBUF signals will remain enabled during hibernate. The internal supply regulator can be woken up by several user-selectable events, all of which are controlled in the VR_CTL register:

- Assertion of the $\overline{\text{RESET}}$ pin always exits hibernate state and requires no modification to VR_CTL. Assertion of $\overline{\text{RESET}}$ will also cause EXT_WAKE to transition high.
- RTC event. Set the wake-up enable control bit (WAKE) to enable wake-up upon an RTC interrupt. This can be any of the RTC interrupts (alarm, daily alarm, day, hour, minute, second, or stopwatch).
- External GP event or Ethernet PHY event. Set the PHY wakeup enable control (PHYWE) bit to enable wakeup upon assertion of the PHY_INT/PG15 pin by an external PHY device. If no external PHY interrupt is needed, set this bit to enable a general-purpose external event via the PG15 pin.
- Activity between the USBDP and USBDM pins. Set the USB wakeup enable control (USBWE) bit to enable wakeup upon detection of USB bus activity on the USBDP/USBDM pins.
- Pin EXT_WAKE is provided to indicate the occurrence of wakeup. EXT_WAKE is an output pin, which is a logical OR of the above wakeup sources, except hardware reset. The pin follows the wakeup signal of the various wakeup sources.

If an external regulator is used to power V_{DDINT} (VRSEL = 1), which is always true on ADSP-BF522/ADSP-BF524/ADSP-BF526 processors, the external regulator can be signaled to shut off V_{DDINT} using the EXT_WAKE signal. Writing b#00 to the FREQ bits of the VR_CTL register, which disables CCLK and SCLK, will also make EXT_WAKE go low. EXT_WAKE will transition

high if any wakeup sources occur or if hardware reset is asserted, which will signal the external voltage regulator to turn V_{DDINT} on again.

When the core is powered down, V_{DDINT} is set to 0 V, and the internal state of the processor is not maintained, with the exception of the VR_CTL register. Therefore, any critical information stored internally (memory contents, register contents, and so on) must be written to a non-volatile storage device prior to removing power. Be sure to set the drive SCKE low during reset control (SCKELOW) bit in VR_CTL to protect against the default reset state behavior of setting the EBIU pins to their inactive state. Failure to set the SCKELOW bit results in the SCKE pin going high during reset, which takes the SDRAM out of self-refresh mode, resulting in data decay in the SDRAM due to loss of refresh rate.

Powering down V_{DDINT} does not affect V_{DDEXT} or V_{DDMEM}. While V_{DDEXT} and V_{DDMEM} are still applied to the processor, external pins are maintained at a three-state level unless specified otherwise.

To power down the internal supply:

1. Write 0 to the appropriate bits in the SIC_IWRx registers to prevent enabled peripheral resources from interrupting the hibernate process.
2. Call the bfrom_SysControl() routine; ensure that the FREQ bits in the VR_CTL variable are set to b#00, and the appropriate wake-up bit(s) to 1 (USB, RTCWAKEUP, or Ethernet Phy). Optionally, set the SCKELOW bit if SDRAM data should be maintained.
3. The bfrom_SysControl() routine executes until V_{DDINT} transitions to 0 V. bfrom_SysControl() never returns.
4. When the processor is woken up, the PLL relocks and the boot sequence defined by the BMODE[3:0] pin settings takes effect.

The WURESET in the SYSCTRL register is set and stays set until the next hardware reset. The WURESET bit may control a conditional boot process.

PLL and VR Registers

The user interface to the PLL and VR registers is through the system control ROM function (bfrom_SysControl()) described in "System Control ROM Function" on page 18-30. The memory-mapped registers (MMRs) are shown in Table 18-8 and illustrated in Figure 18-4 through Figure 18-10.

Failure to utilize the bfrom_SysControl() ROM function to modify PLL and regulator settings may result in processor malfunction.

Table 18-8 shows the functions of the PLL/VR registers.

Table 18-8. PLL/VR Register Mapping

Register Name	Function	Notes	See
PLL_CTL	PLL control register	Requires reprogramming sequence when written	Figure 18-5 on page 18-26
PLL_DIV	PLL divisor register	Can be written freely	Figure 18-4 on page 18-26
PLL_STAT	PLL status register	Monitors active modes of operation	Figure 18-7 on page 18-28
PLL_LOCKCNT	PLL lock count register	Number of CLKINs allowed for PLL to relock	Figure 18-8 on page 18-28
VR_CTL	Voltage regulator control register	Requires PLL reprogramming sequence when written	Figure 18-9 on page 18-29 and Figure 18-10 on page 18-30

PLL_DIV Register

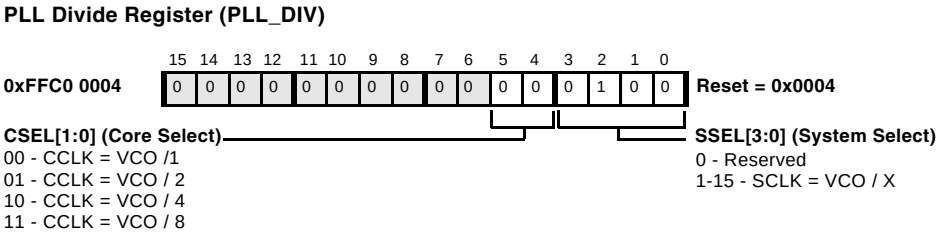

PLL Divide Register (PLL_DIV)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0004
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
Reset = 0x0004
CSEL[1:0] (Core Select)
00 - CCLK = VCO /1
01 - CCLK = VCO / 2
10 - CCLK = VCO / 4
11 - CCLK = VCO / 8
SSEL[3:0] (System Select)
0 - Reserved
1-15 - SCLK = VCO / X

Figure 18-4. PLL Divide Register

PLL_CTL Register

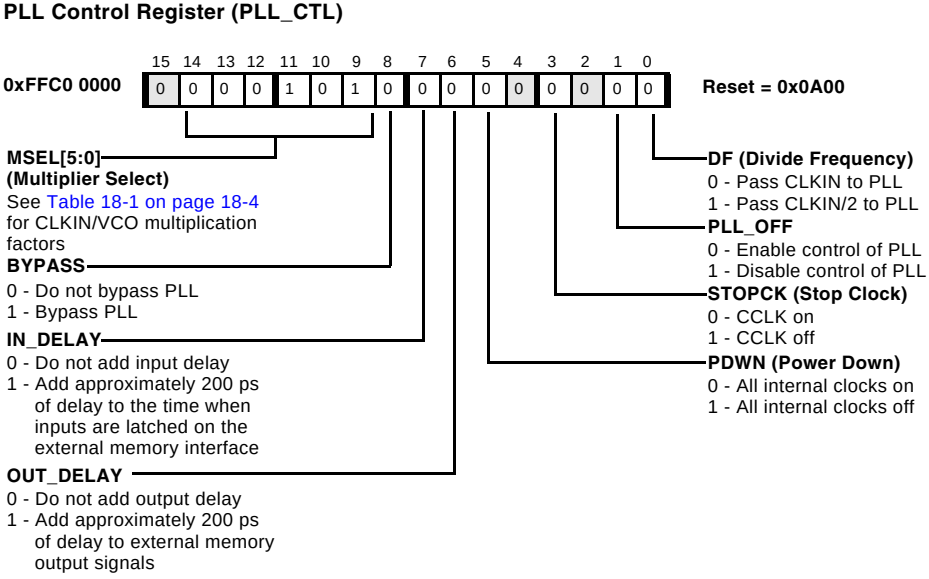

PLL Control Register (PLL_CTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0000
0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0
Reset = 0x0A00
MSEL[5:0]
(Multiplier Select)
See Table 18-1 on page 18-4
for CLKIN/VCO multiplication
factors
BYPASS
0 - Do not bypass PLL
1 - Bypass PLL
IN_DELAY
0 - Do not add input delay
1 - Add approximately 200 ps
of delay to the time when
inputs are latched on the
external memory interface
OUT_DELAY
0 - Do not add output delay
1 - Add approximately 200 ps
of delay to external memory
output signals
DF (Divide Frequency)
0 - Pass CLKIN to PLL
1 - Pass CLKIN/2 to PLL
PLL_OFF
0 - Enable control of PLL
1 - Disable control of PLL
STOPCK (Stop Clock)
0 - CCLK on
1 - CCLK off
PDWN (Power Down)
0 - All internal clocks on
1 - All internal clocks off

Figure 18-5. PLL Control Register for ADSP-BF523/5/7

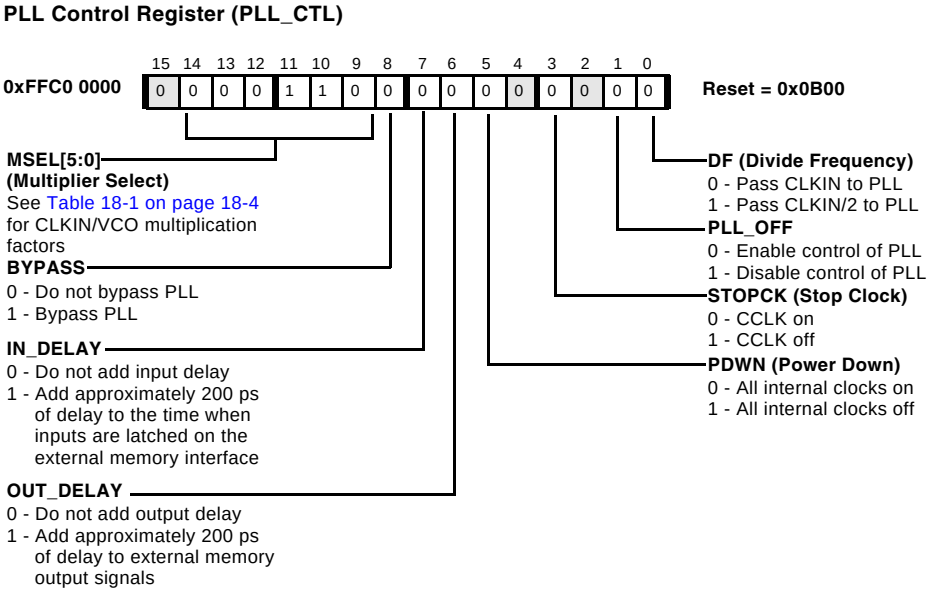

PLL Control Register (PLL_CTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0000
0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
Reset = 0x0B00
MSEL[5:0]
(Multiplier Select)
See Table 18-1 on page 18-4
for CLKIN/VCO multiplication
factors
BYPASS
0 - Do not bypass PLL
1 - Bypass PLL
IN_DELAY
0 - Do not add input delay
1 - Add approximately 200 ps
of delay to the time when
inputs are latched on the
external memory interface
OUT_DELAY
0 - Do not add output delay
1 - Add approximately 200 ps
of delay to external memory
output signals
DF (Divide Frequency)
0 - Pass CLKIN to PLL
1 - Pass CLKIN/2 to PLL
PLL_OFF
0 - Enable control of PLL
1 - Disable control of PLL
STOPCK (Stop Clock)
0 - CCLK on
1 - CCLK off
PDWN (Power Down)
0 - All internal clocks on
1 - All internal clocks off

Figure 18-6. PLL Control Register for ADSP-BFBF522/4/6

PLL_STAT Register

PLL Status Register (PLL_STAT)

Read only. Unless otherwise noted, 1 - Processor operating in this mode. For more information, see “Operating Modes” on page 18-8.

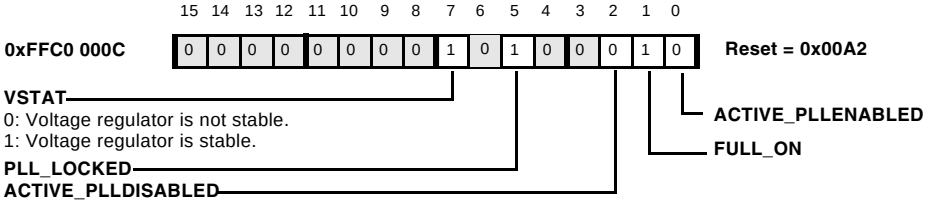

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 000C
0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0
Reset = 0x00A2
VSTAT
0: Voltage regulator is not stable.
1: Voltage regulator is stable.
PLL_LOCKED
ACTIVE_PLLDISABLED
ACTIVE_PLLENABLED
FULL_ON

Figure 18-7. PLL Status Register

PLL_LOCKCNT Register

PLL Lock Count Register (PLL_LOCKCNT)

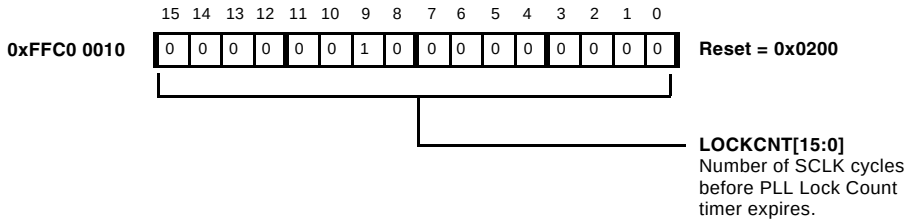

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0010
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
Reset = 0x0200
LOCKCNT[15:0]
Number of SCLK cycles
before PLL Lock Count
timer expires.

Figure 18-8. PLL Lock Count Register

VR_CTL Register

The CLKIN buffer output enable (CLKBUFOE) control bit allows another device, most likely the Ethernet PHY, and the Blackfin processor to run

from a single crystal oscillator. Clearing this bit prevents the CLKBUF pin from driving a buffered version of the input clock CLKIN.

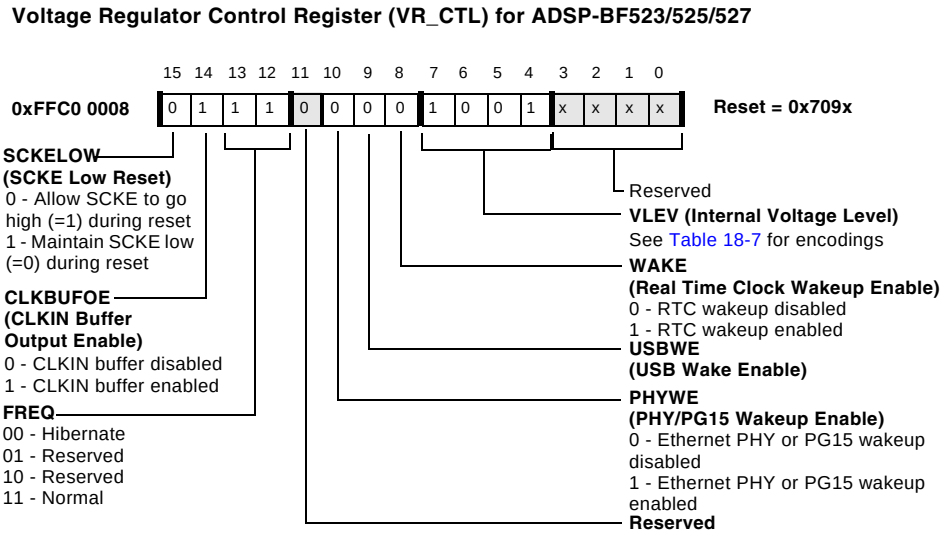

Voltage Regulator Control Register (VR_CTL) for ADSP-BF523/525/527
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0008
0 1 1 1 0 0 0 0 1 0 0 1 x x x x
Reset = 0x709x
SCKELOW
(SCKE Low Reset)
0 - Allow SCKE to go high (=1) during reset
1 - Maintain SCKE low (=0) during reset
CLKBUFOE
(CLKIN Buffer Output Enable)
0 - CLKIN buffer disabled
1 - CLKIN buffer enabled
FREQ
00 - Hibernate
01 - Reserved
10 - Reserved
11 - Normal
Reserved
VLEV (Internal Voltage Level)
See Table 18-7 for encodings
WAKE
(Real Time Clock Wakeup Enable)
0 - RTC wakeup disabled
1 - RTC wakeup enabled
USBWE
(USB Wake Enable)
PHYWE
(PHY/PG15 Wakeup Enable)
0 - Ethernet PHY or PG15 wakeup disabled
1 - Ethernet PHY or PG15 wakeup enabled
Reserved

Figure 18-9. ADSP-BF523/525/527 Voltage Regulator Control Register

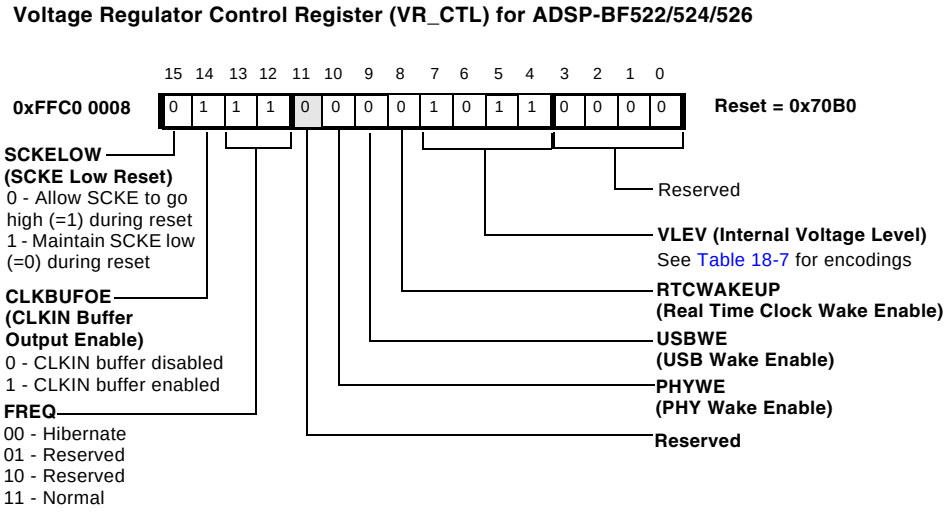

Voltage Regulator Control Register (VR_CTL) for ADSP-BF522/524/526
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0008
0 1 1 1 0 0 0 0 1 0 1 1 0 0 0 0
Reset = 0x70B0
SCKELOW
(SCKE Low Reset)
0 - Allow SCKE to go high (=1) during reset
1 - Maintain SCKE low (=0) during reset
CLKBUFOE
(CLKIN Buffer Output Enable)
0 - CLKIN buffer disabled
1 - CLKIN buffer enabled
FREQ
00 - Hibernate
01 - Reserved
10 - Reserved
11 - Normal
Reserved
VLEV (Internal Voltage Level)
See Table 18-7 for encodings
RTCWAKEUP
(Real Time Clock Wake Enable)
USBWE
(USB Wake Enable)
PHYWE
(PHY Wake Enable)
Reserved

Figure 18-10. ADSP-BF522/524/526 Voltage Regulator Control Register

System Control ROM Function

The PLL and voltage regulator registers should not be accessed directly. Instead, use the `bfrom_SysControl()` function to alter or read the register values. The function resides in the on-chip ROM and can be called by the user following C-language style calling conventions.

Failure to utilize the `bfrom_SysControl()` ROM function to modify PLL and regulator settings may result in processor malfunction.

Entry address: 0xEF00 0038

Arguments:

- `dActionFlags` word in `R0`
- `pSysCtrlSettings` pointer in `R1`
- zero value in `R2`

A potential error message from the internally called `bfrom_OtpRead()` function is forwarded and returned in `R0`.

The system control ROM function does not verify the correctness of the forwarded arguments. Therefore, it is up to the programmer to choose the correct values.

C prototype: `u32 bfrom_SysControl(u32 dActionFlags, ADI_SYSCTRL_VALUES *pSysCtrlSettings, void *reserved);`

The first argument (`u32 dActionFlags`) to the system control ROM function holds the instruction flags. The following flags are supported:

```
#define SYSCTRL_READ          0x00000000
#define SYSCTRL_WRITE         0x00000001
#define SYSCTRL_SYSRESET      0x00000002
#define SYSCTRL_SOFTRESET     0x00000004
#define SYSCTRL_VRCTL         0x00000010
#define SYSCTRL_EXTVOLTAGE    0x00000020
#define SYSCTRL_INTVOLTAGE    0x00000000
#define SYSCTRL_OTPVOLTAGE    0x00000040
#define SYSCTRL_PLLCTL        0x00000100
#define SYSCTRL_PLLDIV        0x00000200
#define SYSCTRL_LOCKCNT       0x00000400
#define SYSCTRL_PLLSTAT       0x00000800
```

With `SYSCTRL_READ` and `SYSCTRL_WRITE`, a read or a write operation is initialized. The `SYSCTRL_SYSRESET` flag performs a system reset, while the `SYSCTRL_SOFTRESET` flag combines a core and system reset. The `SYSCTRL_EXTVOLTAGE` and `SYSCTRL_INTVOLTAGE` flags contain information about whether V_{DDINT} is supplied externally or generated by the on-chip regulator. The `SYSCTRL_OTPVOLTAGE` flag is for factory purposes only. The last five flags (`_VRCTL`, `_PLLCTL`, `_PLLDIV`, `_LOCKCNT`, `_PLLSTAT`) tells the system control ROM function which registers are to be written to or read from. Note that the `SYSCTRL_PLLSTAT` flag is read-only.

The second argument (`ADI_SYSCTRL_VALUES *pSysCtrlSettings`) to the system control ROM function passes a pointer to a special structure, which has entries for all PLL and voltage regulator registers. It is predefined in the `bfrom.h` header file as follows.

```
typedef struct
{
   u16 uwVrCtl;
   u16 uwPllCtl;
   u16 uwPllDiv;
   u16 uwPllLockCnt;
   u16 uwPllStat;
}
   ADI_SYSCTRL_VALUES;
```

The third argument to the system control ROM function is reserved and should be kept zero (NULL pointer).

The function's return value is described in the following `bfrom_OtpRead()` ROM routine descriptions; whereby a single-bit warning is suppressed.

The system control ROM function executes the correct steps and programming sequence for the Dynamic Power Management System of the Blackfin processor.

Programming Model

The programming model for the system control ROM function in C/C++ and Assembly is described in the following sections.

Accessing the System Control ROM Function in C/C++

To read the `PLL_DIV` and `PLL_CTL` register values, for example, specify the `SYSCTRL_READ` instruction flag along with the `SYSCTRL_PLLCTL` and

SYSCTRL_PLLDIV register flags. The bfrom_OtpRead() function then only updates the uwPllCtl and uwPllDiv variables:

```
ADI_SYSCTRL_VALUES read;
bfrom_SysControl (SYSCTRL_READ | SYSCTRL_PLLCTL | SYSCTRL_PLLDIV,
&read, NULL)
;
```

The read.uwPllCtl and read.uwPllDiv variables access the PLL_CTL and PLL_DIV register values, respectively. To update the register values, specify the SYSCTRL_WRITE instruction flag along with the register flags of those registers that should be modified and have valid data in the respective ADI_SYSCTRL_VALUES variables:

```
ADI_SYSCTRL_VALUES write;
write.uwPllCtl = 0x1400;
write.uwPllDiv = 0x0005;
bfrom_SysControl (SYSCTRL_WRITE | SYSCTRL_PLLCTL |SYSCTRL_PLLDIV,
&write, NULL);
```

Accessing the System Control ROM Function in Assembly

The assembler supports C structs, which is required to import the file bfrom.h:

```
#include <bfrom.h>
.IMPORT "bfrom.h";
.STRUCT ADI_SYSCTRL_VALUES dpm;
```

You can pre-load the struct:

```
.STRUCT ADI_SYSCTRL_VALUES dpm = { 0x40DB, 0x1400, 0x0005,
0x0200, 0x00A2 };
```

or load the values dynamically inside the code:

```
P5.H = hi(dpm);
P5.L = lo(dpm->uwVrCtl);
R7 = 0x40DB (z);
```

```
w[P5] = R7;
P5.L = lo(dpm->uwPllCtl);
R7 = 0x1400 (z);
w[P5] = R7;
P5.L = lo(dpm->uwPllDiv);
R7 = 0x0005 (z);
w[P5] = R7;
P5.L = lo(dpm->uwPllLockCnt);
R7 = 0x0200 (z);
w[P5] = R0;
```

The function `u32 bfrom_SysControl(u32 dActionFlags, ADI_SYSCTRL_VALUES *pSysCtrlSettings, void *reserved);` can be accessed by `bfrom_SysControl`. Following the C/C++ run-time environment conventions, the parameters passed are hold by the data registers `R0`, `R1`, and `R2`.

```
/* 10 = sizeof(ADI_SYSCTRL_VALUES). uimm18m4: 18-bit unsigned
field that must be a multiple of 4, with a range of 8 through
262,152 bytes (0x00000 through 0x3FFFC) */
link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
/* Allocate at least 12 bytes on the stack for outgoing argu-
ments, even if the function being called requires less than this.
*/
SP += -12;
R0 = SYSCTRL_WRITE        |
   SYSCTRL_VRCTL          |
   SYSCTRL_INTVOLTAGE     |
   SYSCTRL_PLLCTL         |
   SYSCTRL_PLLDIV;
R1.H = hi(dpm);
R1.L = lo(dpm);
R2 = 0 (z);
P5.H = hi(BFROM_SYSCONTROL);
P5.L = lo(BFROM_SYSCONTROL);
```

```
call(P5);
SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
```

The processor's internal scratchpad memory can be used as an alternative for taking a C struct. Therefore, the stack/frame pointer must be loaded and passed.

```
/* 10 = sizeof(ADI_SYSCTRL_VALUES). uimm18m4: 18-bit unsigned
field that must be a multiple of 4, with a range of 8 through
262,152 bytes (0x00000 through 0x3FFFC) */
link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
/* Allocate at least 12 bytes on the stack for outgoing argu-
ments, even if the function being called requires less than this.
*/
SP += -12;
R7 = 0;
R7.L = 0x40DB;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+off-
setof(ADI_SYSCTRL_VALUES,uwVrCtl)] = R7;
R7.L = 0x1400;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+off-
setof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R7;
R7.L = 0x0005;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+off-
setof(ADI_SYSCTRL_VALUES,uwPllDiv)] = R7;
R7.L = 0x0200;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+off-
setof(ADI_SYSCTRL_VALUES,uwPllLockCnt)] = R7;
R0 = SYSCTRL_WRITE     |
   SYSCTRL_VRCTL       |
   SYSCTRL_INTVOLTAGE  |
   SYSCTRL_PLLCTL
```

```
   SYSCTRL_PLLDIV   ;
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0;
P5.H = hi(BFROM_SYSCONTROL);
P5.L = lo(BFROM_SYSCONTROL);
call(P5);
SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
```

Programming Examples

The following code examples illustrate how to use the system control ROM function to effect various operating mode transitions.

The following examples are only meant to demonstrate how to program the PLL registers. Do not assume that the voltages and frequencies shown in the examples are supported by your processor. Instead, check your product's data sheet for supported voltages and frequencies.

Some setup code has been removed for clarity, and the following assumptions are made.

- PLL control (`PLL_CTL`) register setting: 0x0A00
- PLL divider (`PLL_DIV`) register setting: 0x0004
- PLL lock count (`PLL_LOCKCNT`) register setting: 0x0200
- Clock in (`CLKIN`) frequency: 25 MHz

VCO frequency is 125 MHz, core clock frequency is 125 MHz, and system clock frequency is 31.25 MHz.

- Voltage regulator control (VR_CTL) register setting: 0x709x

Logical voltage level (V_{DDINT}) is at 1.10 V

For operating mode transition and voltage regulator examples:

- **C**

```
#include  <blackfin.h>

#include <bfrom.h>
```

- **Assembly**

```
#include  <blackfin.h>

#include <bfrom.h>

.IMPORT "bfrom.h";

#define IMM32(reg,val) reg##.H=hi(val);

reg##.L=lo(val);
```

Full-on Mode to Active Mode and Back

Listing 18-1 and Listing 18-2 provide code for transitioning from the full-on operating mode to active mode in C and Blackfin assembly code, respectively.

Listing 18-1. Transitioning from Full-on Mode to Active Mode (C)

```
void active(void)
{
ADI_SYSCTRL_VALUES active;
```

```
bfrom_SysControl(SYSCTRL_READ | SYSCTRL_INTVOLTAGE |
SYSCTRL_PLLCTL, &active, NULL);
active.uwPllCtl |= (BYPASS | PLL_OFF); /* PLL_OFF bit optional */
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_INTVOLTAGE |
SYSCTRL_PLLCTL, &active, NULL);|
return;
}
```

Listing 18-2. Transitioning from Full-on Mode to Active Mode (ASM)

```
__active:
link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;
R0 = (SYSCTRL_READ | SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
R0 = w[FP+-sizeof(ADI_SYSCTRL_VALUES) + offset of
(ADI_SYSCTRL_VALUES,uwPllCtl)];
bitset(R0,bitpos(BYPASS));
bitset(R0,bitpos(PLL_OFF)); /* optional */
w[FP+-sizeof(ADI_SYSCTRL_VALUES) + offset of
(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;
R0 = (SYSCTRL_WRITE | SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
SP += 12;
(R7:0,P5:0) = [SP++];
```

```
unlink;
rts;
__active.end:
```

To return from active mode (go back to full-on mode), the BYPASS bit and the PLL_OFF bit must be cleared again, respectively.

Transition to Sleep Mode or Deep Sleep Mode

Listing 18-3 and Listing 18-4 provide code for transitioning from the full-on operating mode to sleep or deep sleep mode in C and Blackfin assembly code, respectively.

Listing 18-3. Transitioning to Sleep Mode or Deep Sleep Mode (C)

```
void sleep(void)
{
ADI_SYSCTRL_VALUES sleep;
bfrom_SysControl(SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_READ, &sleep, NULL);
sleep.uwPllCtl |= STOPCK;    /* either: Sleep Mode */
sleep.uwPllCtl |= PDWN;      /* or: Deep Sleep Mode */
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_INTVOLTAGE |
SYSCTRL_PLLCTL, &sleep, NULL);
return;
}
```

Listing 18-4. Transitioning to Sleep Mode or Deep Sleep Mode (ASM)

```
__sleep:
link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;
R0 = (SYSCTRL_READ | SYSCTRL_INTVOLTAGE |
```

```
SYSCTRL_PLLCTL);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
R0 = w[FP+-sizeof(ADI_SYSCTRL_VALUES) + offset of
(ADI_SYSCTRL_VALUES,uwPllCtl)];
bitset(R0,bitpos(STOPCK)); /* either: Sleep Mode */
bitset(R0,bitpos(PDWN)); /* or: Deep Sleep Mode */
w[FP+-sizeof(ADI_SYSCTRL_VALUES) + offset of
(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;
R0 = (SYSCTRL_WRITE | SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
__sleep.end:
```

Set Wakeups and Entering Hibernate State

Listing 18-5 and Listing 18-6 provide code for configuring the regulator wakeups (RTC wakeup) and placing the regulator in the hibernate state in C and Blackfin assembly code, respectively.

Listing 18-5. Configuring Regulator Wakeups and Entering Hibernate State (C)

```
void hibernate(void)
```

```
{
ADI_SYSCTRL_VALUES hibernate;
/* SCLKELOW = 1: Enable Drive SCKE Low During Reset */
/* Protect SDRAM contents during reset after wakeup */
hibernate.uwVrCtl=SCKELOW |
   WAKE | /* RTC/Reset Wake-Up Enable */
HIBERNATE ; /* Powerdown/Bypass On-Board Regulation */
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_VRCTL |
SYSCTRL_INTVOLTAGE, &hibernate, NULL);
/*Hibernate State: no code executes until wakeup triggers reset*/
}
```

Listing 18-6. Configuring Regulator Wakeups and Entering Hibernate State (ASM)

```
__hibernate:
link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;
cli R6; /* disable interrupts, copy IMASK to R6 */
/* SCLKELOW = 1: Enable Drive SCKE Low During Reset */
/* Protect DDR contents during reset after wakeup */
R0.L =  SCKELOW   |
    WAKE      | /* RTC/Reset Wake-Up Enable */
    HIBERNATE ; /* Powerdown/Bypass On-Board Regulation */
w[FP+-sizeof(ADI_SYSCTRL_VALUES) + offset of
(ADI_SYSCTRL_VALUES,uwVrCtl)] = R0;
R0 = (SYSCTRL_WRITE | SYSCTRL_VRCTL | SYSCTRL_INTVOLTAGE);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
```

```
/*Hibernate State: no code executes until wakeup triggers reset*/
__hibernate.end:
```

Perform a System Reset or Soft-Reset

Listing 18-7 and Listing 18-8 provide code for executing a system reset *or* a soft-reset (system and core reset) in C and Blackfin assembly code, respectively.

Listing 18-7. Execute a System Reset or a Soft-Reset (C)

```
void reset(void)
{
bfrom_SysControl(SYSCTRL_SYSRESET, NULL, NULL); /* either */
bfrom_SysControl(SYSCTRL_SOFTRESET, NULL, NULL); /* or */
return;
}
```

Listing 18-8. Execute a System Reset or a Soft-Reset (ASM)

```
__reset:
link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;
R0 = SYSCTRL_SYSRESET; /* either */
R0 = SYSCTRL_SOFTRESET; /* or */
R1 = 0 (z);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
```

```
rts;
__reset.end:
```

In Full-on Mode, Change VCO Frequency, Core Clock Frequency, and System Clock Frequency

Listing 18-9 and Listing 18-10 provide C and Blackfin assembly code for changing the CLKIN to VCO multiplier (from 10x to 21x), keeping the CSEL divider at 1, and changing the SSEL divider (from 5 to 4) in the full-on operating mode.

Listing 18-9. Transition of Frequencies (C)

```
void frequency(void)
{
ADI_SYSCTRL_VALUES frequency;
/* Set MSEL = 0-63 --> VCO = CLKIN*MSEL */
frequency.uwPllCtl = SET_MSEL(21) ;
/* Set SSEL = 1-15 --> SCLK = VCO/SSEL */
/* CCLK = VCO / 1 */
frequency.uwPllDiv = SET_SSEL(4) | CSEL_DIV1;
frequency.uwPllLockCnt = 0x0200;
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_INTVOLTAGE |
SYSCTRL_PLLCTL | SYSCTRL_PLLDIV | SYSCTRL_LOCKCNT |, &frequency,
NULL);
return;
}
```

Listing 18-10. Transition of Frequencies (ASM)

```
__frequency:
link sizeof(ADI_SYSCTRL_VALUES)+2;
```

```
[--SP] = (R7:0,P5:0);
SP += -12;
/* write the struct */
R0 = 0;
R0.L = SET_MSEL(21) ; /* Set MSEL = 0-63 --> VCO = CLKIN*MSEL */
w[FP+-sizeof(ADI_SYSCTRL_VALUES) + offset of
(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;
R0.L = SET_SSEL(4) | /* Set SSEL = 1-15 --> SCLK = VCO/SSEL */
   CSEL_DIV1 ;   /* CCLK = VCO/1 */
w[FP+-sizeof(ADI_SYSCTRL_VALUES) + offset of
(ADI_SYSCTRL_VALUES,uwPllDiv)] = R0;
R0.L = 0x0200;
w[FP+-sizeof(ADI_SYSCTRL_VALUES) + offset of
(ADI_SYSCTRL_VALUES,uwPllLockCnt)] = R0;
/* argument 1 in R0 */
R0 = (SYSCTRL_WRITE | SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_PLLDIV);
/* argument 2 in R1: structure lays on local stack */
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
/* argument 3 must always be NULL */
R2 = 0;
/* call of SysControl function */
IMM32(P4,BFROM_SYSCONTROL);
call (P4); /* R0 contains the result from SysControl */
SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
__frequency.end:
```

Changing Voltage Levels

Listing 18-11 and Listing 18-12 provide code for changing the voltage level dynamically, in C and Blackfin assembly code, respectively. The voltage level is increased to 1.20 V. Additional code may be required to alter the core clock frequency when voltage level is being decreased. Refer to the processor data sheet for the applicable VLEV voltage range and associated supported core clock speeds.

Listing 18-11. Changing Core Voltage via the On-chip Regulator (C)

```
void voltage(void)
{
ADI_SYSCTRL_VALUES voltage;
voltage.uwVrCtl = VLEV_120  | /* VLEV = 1.20 V */
CLKBUFOE  | /* CLKIN Buffer Output Enable */
FREQ ; /* Switching Oscillator Frequency for Regulator */
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_VRCTL |
SYSCTRL_INTVOLTAGE, &voltage, NULL);
return;
}
```

Listing 18-12. Changing Core Voltage via the On-chip Regulator (ASM)

```
__voltage:
link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;
R0.L = VLEV_120 | /* VLEV = 1.20 V */
    CLKBUFOE | /* CLKIN Buffer Output Enable */
w[FP+-sizeof(ADI_SYSCTRL_VALUES) + offset of
(ADI_SYSCTRL_VALUES,uwVrCtl)] = R0;
R0 = (SYSCTRL_WRITE | SYSCTRL_VRCTL | SYSCTRL_INTVOLTAGE);
R1 = FP;
```

```
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
__voltage.end:
```

The previous sequence must also be executed when the V_{DDINT} voltage is applied externally to ensure internal timings can appropriately adjusted for the constant or changing V_{DDINT} voltage. In this case replace the `SYSCTRL_INTVOLTAGE` flag by the `SYSCTRL_EXTVOLTAGE` flag.

Listing 18-13 provides C code for changing the voltage level dynamically. The user must include his own code for accessing the external voltage regulator.

Listing 18-13. Changing Core Voltage on ADSP-BF522/ADSP-BF524/ADSP-BF526 (C)

```
void voltage(void)
{
ADI_SYSCTRL_VALUES voltage;
u32 ulCnt = 0;
bfrom_SysControl( SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_READ, &init, NULL );
init.uwPllCtl |= BYPASS;
init.uwPllLockCnt = 0x0200;
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_PLLCTL | SYSCTRL_LOCKCNT
| SYSCTRL_EXTVOLTAGE, &voltage, NULL);
/* Put your code for accessing the external voltage regulator
here */
```

```
/* A delay loop is required to ensure VDDint is stable and the
PLL has re-locked. As this is depending on the external voltage
regulator circuitry the user must ensure timings are kept. The
compiler (no optimization enabled) will create a loop that takes
about 10 cycles. Time base is CLKIN as the PLL is bypassed. We
need 0x0200 CLKIN cycles that represent PLL_LOCKCNT and addition-
ally the time required by the circuitry */
ulCnt = 0x0200 + 0x0200;
while (ulCnt != 0) {ulCnt--;}
init.uwPllCtl &= ~BYPASS;
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_PLLCTL |
SYSCTRL_EXTVOLTAGE, &voltage, NULL);
return;
}
```

19 SYSTEM DESIGN

This chapter provides hardware, software and system design information to aid users in developing systems based on the Blackfin processor. The design options implemented in a system are influenced by cost, performance, and system requirements. In many cases, design issues cited here are discussed in detail in other sections of this manual. In such cases, a reference appears to the corresponding section of the text, instead of repeating the discussion in this chapter.

Pin Descriptions

Refer to the processor data sheet for pin information, including pin numbers for the 208-ball Pb-free sparse MBGA and 289-ball MBGA.

Managing Clocks

Systems can drive the clock inputs with a crystal oscillator or a buffered, shaped clock derived from an external clock oscillator. The external clock connects to the processor's `CLKIN` pin. It is not possible to halt, change, or operate `CLKIN` below the specified frequency during normal operation. The processor uses the clock input (`CLKIN`) to generate on-chip clocks. These include the core clock (`CCLK`) and the peripheral clock (`SCLK`).

Managing Core and System Clocks

The processor produces a multiplication of the clock input provided on the CLKIN pin to generate the PLL VCO clock. This VCO clock is divided to produce the core clock (CCLK) and the system clock (SCLK). The core clock is based on a divider ratio that is programmed via the CSEL bit settings in the PLL_DIV register. The system clock is based on a divider ratio that is programmed via the SSEL bit settings in the PLL_DIV register. For detailed information about how to set and change CCLK and SCLK frequencies, see "System Reset and Booting" on page 17-1.

Configuring and Servicing Interrupts

A variety of interrupts are available. They include both core and peripheral interrupts. The processor assigns default core priorities to system-level interrupts. However, these system interrupts can be remapped via the system interrupt assignment registers (SIC_IARx). For more information, see Chapter 5, "System Interrupts".

The processor core supports nested and non-nested interrupts, as well as self-nested interrupts. For explanations of the various modes of servicing events, please see the Blackfin Processor Programming Reference.

Semaphores

Semaphores provide a mechanism for communication between multiple processors or processes/threads running in the same system. They are used to coordinate resource sharing. For instance, if a process is using a particular resource and another process requires that same resource, it must wait until the first process signals that it is no longer using the resource. This signalling is accomplished via semaphores.

Semaphore coherency is guaranteed by using the test and set byte (atomic) instruction (TESTSET). The TESTSET instruction performs these functions.

- Loads the half word at memory location pointed to by a P-register. The P-register must be aligned on a half-word boundary.
- Sets CC if the value is equal to zero.
- Stores the value back in its original location (but with the most significant bit (MSB) of the low byte set to 1).

The events triggered by TESTSET are atomic operations. The bus for the memory where the address is located is acquired and not relinquished until the store operation completes. In multithreaded systems, the TESTSET instruction is required to maintain semaphore consistency.

To ensure that the store operation is flushed through any store or write buffers, issue an SSYNC instruction immediately after semaphore release.

The TESTSET instruction can be used to implement binary semaphores or any other type of mutual exclusion method. The TESTSET instruction supports a system-level requirement for a multicycle bus lock mechanism.

The processor restricts use of the TESTSET instruction to the external memory region only. Use of the TESTSET instruction to address any other area of the memory map may result in unreliable behavior.

Example Code for Query Semaphore

Listing 17-1 provides an example of a query semaphore that checks the availability of a shared resource.

Listing 19-1. Query Semaphore

```
/* Query semaphore. Denotes “Busy” if its value is nonzero. Wait
until free (or reschedule thread-- see note below). P0 holds
address of semaphore. */
```

```
QUERY:
TESTSET ( P0 ) ;
IF !CC JUMP QUERY ;
/* At this point, semaphore has been granted to current thread,
and all other contending threads are postponed because semaphore
value at [P0] is nonzero. Current thread could write thread_id to
semaphore location to indicate current owner of resource. */
R0.L = THREAD_ID ;
B[P0] = R0 ;
/* When done using shared resource, write a zero byte to [P0] */
R0 = 0 ;
B[P0] = R0 ;
SSYNC ;
/* NOTE: Instead of busy idling in the QUERY loop, one can use an
operating system call to reschedule the current thread. */
```

Data Delays, Latencies and Throughput

For detailed information on latencies and performance estimates on the DMA and external memory buses, refer to Chapter 2, “Chip Bus Hierarchy”.

Bus Priorities

For an explanation of prioritization between the various internal buses, refer to Chapter 2, “Chip Bus Hierarchy”.

External Memory Design Issues

This section describes design issues related to external memory.

Example Asynchronous Memory Interfaces

This section shows glueless connections to 16-bit wide SRAM. Note this interface does not require external assertion of ARDY, since the internal wait state counter is sufficient for deterministic access times of memories.

Figure 17-1 shows the interface to 8-bit SRAM or flash. Figure 17-2 shows the interface to 16-bit SRAM or flash

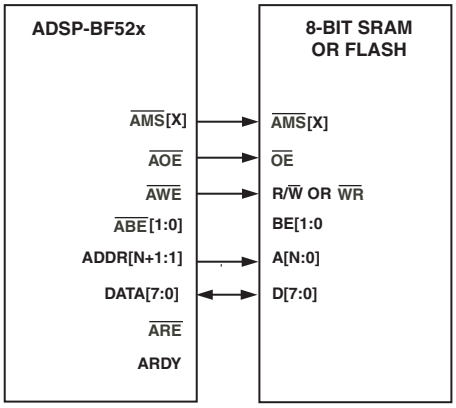


Figure 19-1. Interface to 8-Bit SRAM or Flash

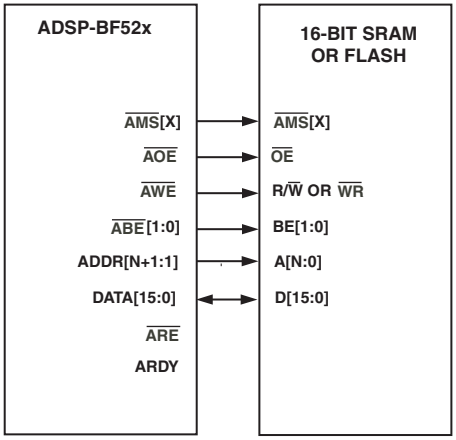


Figure 19-2. Interface to 16-Bit SRAM or Flash

Figure 17-3 shows the system interconnect required to support 16-bit memories. Note this application requires the 16-bit packing mode be enabled for this bank of memory. Otherwise, the programming model must ensure that every other 16-bit memory location is accessed starting on an even (byte address[1:0] = 00) 16-bit address.

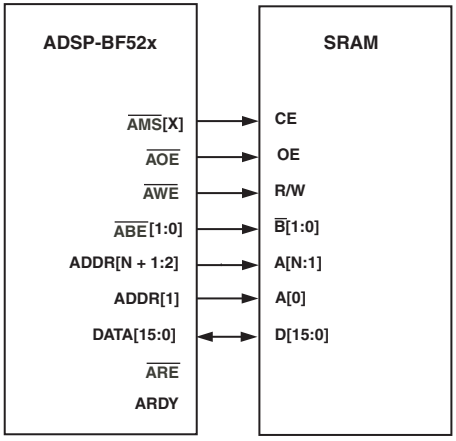


Figure 19-3. Interface to 16-Bit SRAM

Avoiding Bus Contention

Because the three-stated data bus is shared by multiple devices in a system, be careful to avoid contention. Contention causes excessive power dissipation and can lead to device failure. Contention occurs during the time one device is getting off the bus and another is getting on. If the first device is slow to three-state and the second device is quick to drive, the devices contend.

There are two cases where contention can occur. The first case is a read followed by a write to the same memory space. In this case, the data bus drivers can potentially contend with those of the memory device addressed by the read. The second case is back-to-back reads from two different memory spaces. In this case, the two memory devices addressed by the two reads can potentially contend at the transition between the two read operations.

To avoid contention, program the turnaround time (bank transition time) appropriately in the asynchronous memory bank control registers. This feature allows software to set the number of clock cycles between these types of accesses on a bank-by-bank basis. Minimally, the external bus interface unit (EBIU) provides one cycle for the transition to occur.

High-Frequency Design Considerations

Because the processor can operate at very fast clock frequencies, signal integrity and noise problems must be considered for circuit board design and layout. The following sections discuss these topics and suggest various techniques to use when designing and debugging signal processing systems.

Signal Integrity

In addition to reducing signal length and capacitive loading, critical signals should be treated like transmission lines.

Capacitive loading and signal length of buses can be reduced by using a buffer for devices that operate with wait states (for example, SDRAMs). This reduces the capacitance on signals tied to the zero-wait-state devices, allowing these signals to switch faster and reducing noise-producing current spikes. Extra care should be taken with certain signals such as external memory, read, write, and acknowledge strobes.

Use simple signal integrity methods to prevent transmission line reflections that may cause extraneous extra clock and sync signals. Additionally, avoid overshoot and undershoot that can cause long term damage to input pins.

Some signals are especially critical for short trace length and usually require series termination. The CLKIN pin should have impedance matching series resistance at its driver. SPORT interface signals TCLK, RCLK, RFS, and TFS should use some termination. Although the serial ports may be operated at a slow rate, the output drivers still have fast edge rates and for longer distances the drivers often require resistive termination located at the source. (Note also that TFS and RFS should not be shorted in multi-channel mode.) On the PPI interface, the PPI_CLK and SYNC signals also benefit from these standard signal integrity techniques. If these pins have multiple sources, it will be difficult to keep the traces short. Consider termination of SDRAM clocks, control, address, and data to improve signal quality and reduce unwanted EMI.

Adding termination to fix a problem on an existing board requires delays for new artwork and new boards. A transmission line simulator is recommended for critical signals. IBIS models are available from Analog Devices Inc. that will assist signal simulation software. Some signals can be corrected with a small zero or 22 ohm resistor located near the driver. The resistor value can be adjusted after measuring the signal at all endpoints.

For details, see the reference sources in “Recommended Reading” on page 17-13 for suggestions on transmission line termination.

Other recommendations and suggestions to promote signal integrity:

- Use more than one ground plane on the Printed Circuit Board (PCB) to reduce crosstalk. Be sure to use lots of vias between the ground planes.
- Keep critical signals such as clocks, strobes, and bus requests on a signal layer next to a ground plane and away from or laid out perpendicular to other non-critical signals to reduce crosstalk.
- Experiment with the board and isolate crosstalk and noise issues from reflection issues. This can be done by driving a signal wire from a pulse generator and studying the reflections while other components and signals are passive.

Decoupling Capacitors and Ground Planes

Ground planes must be used for the ground and power supplies. The capacitors should be placed very close to the VDDEXT and VDDINT pins of the package as shown in Figure 17-4. Use short and fat traces for this. The ground end of the capacitors should be tied directly to the ground plane inside the package footprint of the processor (underneath it, on the bottom of the board), not outside the footprint. A surface-mount capacitor is recommended because of its lower series inductance.

Connect the power plane to the power supply pins directly with minimum trace length. A ground plane should be located near the component side of the board to reduce the distance that ground current must travel through vias. The ground planes must not be densely perforated with vias or traces as their effectiveness is reduced.

VDDINT is the highest frequency and requires special attention. Two things help power filtering above 100 MHz. First, capacitors should be physically

small to reduce the inductance. Surface mount capacitors of size 0402 give better results than larger sizes. Secondly, lower values of capacitance will raise the resonant frequency of the LC circuit. While a cluster of 0.1µF is acceptable below 50 MHz, a mix of 0.1µF, 0.01µF, 0.001µF and even 100 pF is preferred in the 500 MHz range.

Note that the instantaneous voltage on both internal and external power pins must at all times be within the recommended operating conditions as specified in the product data sheet. Local "bulk capacitance" (many microfarads) is also necessary. Although all capacitors should be kept close to the power consuming device, small capacitance values should be the closest and larger values may be placed further from the chip.

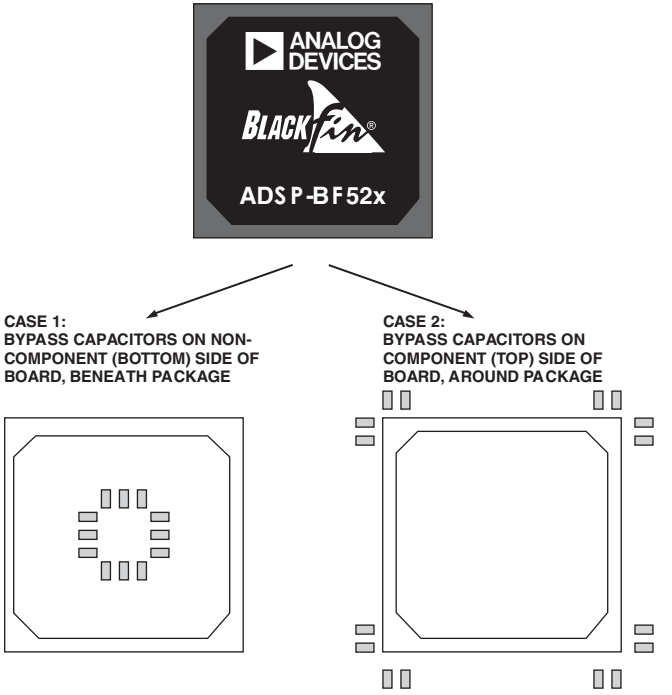

ANALOG DEVICES
BLACKfin
ADSP-BF52x
CASE 1:
BYPASS CAPACITORS ON NON-COMPONENT (BOTTOM) SIDE OF BOARD, BENEATH PACKAGE
CASE 2:
BYPASS CAPACITORS ON COMPONENT (TOP) SIDE OF BOARD, AROUND PACKAGE

Figure 19-4. Bypass Capacitor Placement

5 Volt Tolerance

Outputs that connect to inputs on 5 V devices can float or be pulled up to 5 V. Most Blackfin pins are not 5 V tolerant. There are a few exceptions such as the TWI pins. Level shifters are required on all other Blackfin pins to keep the pin voltage at or below absolute maximum ratings.

Test Point Access

The debug process is aided by test points on signals such as CLKOUT or SCLK, bank selects, PPICLK, and $\overline{\text{RESET}}$. If selection pins such as boot mode are connected directly to power or ground, they are inaccessible under a BGA chip. Use pull-up and pull-down resistors instead.

Oscilloscope Probes

When making high-speed measurements, be sure to use a “bayonet” type or similarly short (< 0.5 inch) ground clip, attached to the tip of the oscilloscope probe. The probe should be a low-capacitance active probe with 3 pF or less of loading. The use of a standard ground clip with 4 inches of ground lead causes ringing to be seen on the displayed trace and makes the signal appear to have excessive overshoot and undershoot. To see the signals accurately, a 1 GHz or better sampling oscilloscope is needed.

Recommended Reading

For more information, refer to *High-Speed Digital Design: A Handbook of Black Magic*, Johnson & Graham, Prentice Hall, Inc., ISBN 0-13-395724-1.

This book is a technical reference that covers the problems encountered in state of the art, high-frequency digital circuit design. It is an excellent

source of information and practical ideas. Topics covered in the book include:

- High-speed properties of logic gates
- Measurement techniques
- Transmission lines
- Ground planes and layer stacking
- Terminations
- Vias
- Power systems
- Connectors
- Ribbon cables
- Clock distribution
- Clock oscillators

Consult your CAD software tools vendor. Some companies offer demonstration versions of signal integrity software. Simply by using their free software, you can learn:

- Transmission lines are real
- Unterminated printed circuit board traces will ring and have overshoot and undershoot
- Simple termination will control signal integrity problems

Resetting the Processor

The reset pin requires a monotonic rise and fall. Therefore the pin should not be connected directly to an R/C time delay because such a circuit could be noise sensitive. In addition to the hardware reset mode provided via the RESET pin, the processor supports several software reset modes. For detailed information on the various modes, see Blackfin Processor Programming Reference. The processor state after reset is also described in the programming reference.

Recommendations for Unused Pins

Most often, there is no need to terminate unused pins, but the handful that do require termination are listed at the end of the pin list description section of the product data sheet.

If the real-time clock is not used, RTXI should be pulled low.

Also note that unused peripherals may have separate power connections. These should be driven to the specified value.

Programmable Outputs

During power up, each GPIO pin is set to an input and any pins used in the system as an output should be connected to a pullup or pulldown resistor to maintain the desired state.

This would be particularly important in motor drive applications. It is also important for UART TX and RTS, SPI and serial TWI, or other communications interfaces. Some memory enable pull-ups may also be desired.

After the boot cycle, each GPIO pin may be set to input or output depending on ADSP-BF52x model number and the boot cycle chosen. The I/O / GPIO muxing of all pins may need to be reprogrammed to sup-

port the users application. Care should be taken for compatibility of function and state, before boot, during boot, and application pin usage.

USB System Hardware Design

The UTMI (Universal Transceiver Macro Interface) of the USB controller is unique. It is sometimes called the PHY section of the USB controller. Its features allow direct connection to a USB OTG cable connector. System hardware requirements are discussed below.

The UTMI section of the USB does not use the system clock. An external clock is needed. The default value would be a 24 MHz clock to the USB_XI pin or a 24 MHz crystal circuit used with USB_XI and USB_XO. If using a crystal, use the same circuit as shown in the datasheet for CLKIN and XTAL. Other frequencies can be used to generate the internal clock of exactly 960 MHz. Use the formula:
960 MHz = frequency of USB_XI × 2 × m where m is an integer.

The UTMI section of the USB has the standard level of ESD protection for integrated circuits. External protection diodes should be added near the connector for DP, DM, ID and VBUS. There are several sources of ESD protection designed specifically for USB2.

When operating in USB OTG host mode, the user must supply an external 5 volt supply at 8 ma or more. The 5 volts can be provided with a "charge pump" or a normal voltage regulator depending on the available input voltages available for the application. In either case the 5 volt supply must be enabled and disabled in software using a GPIO with a resistor to set the initial value to disable the external 5 volt source.

DP and DM are intended for direct connection to the D+ and D- of a USB cable connector. They do not require any pullup or pulldown resistors as these are applied internally by the UTMI in accordance with the programmed application mode. Note also that like any USB design, DP and DM

should be routed as a differential pair with 90 to 100 ohms mutual impedance.

If using the USB in device mode only, you may put a pullup resistor on USB_ID pin or leave the pin disconnected. Either pullup or disconnected will work. The USB_RSET pin can be left open.

Voltage Regulator System Hardware Design

An internal voltage regulator can be used with the recommended external circuit to provide a flexible system of power management. Many applications will only require a fixed internal voltage value and can use a simple external voltage regulator to generate the V_{DDINT} supply voltage. The voltage regulator has two modes set by the VRSEL pin; normal pulse width control of an external FET and external supply mode so that it can signal a power down during hibernate to an external regulator. VRSEL is high for external regulators and low for internal regulator applications. In this external mode VROUT becomes EXT_WAKE. If the internal regulator is used, another EXT_WAKE pin can control other power sources in the system during the hibernate operating mode. Both signals are high true for power-up and may be connected directly to the low true shut down input of many common regulators.

The (Soft Start / Power Good indication) pin is a new feature in ADSP-BF52x. A summary of its operation is given in Table 19-1. The

mode of this pin changes from SS to $\overline{\text{PG}}$ according to the state of the VRSEL pin.

Table 19-1. SS/$\overline{\text{PG}}$ Options

Internal Regulator VRSEL pin = 0 Function: SOFT START	**External Regulator VRSEL pin = 1 Function: NOT POWER GOOD**
OPEN for no Soft Start	GROUND if no hibernation considerations
1 nF to GND for 100 µs ramp	Power Good from regulator circuit (often requires inverter)
5 nF to GND for 500 µs ramp	Voltage divider set to less than operating V_{DDINT} voltage

For VRSEL = Logic 0, Internal Regulator, SS Mode

If using the internal regulator with an external FET switch, the pin should be unconnected or have a capacitor to ground for SS (soft start) mode. At power-up time, the ramp speed of V_{DDINT} may or may not be limited by the ramp time of V_{DDEXT}. When changing voltage or when waking up from the hibernate state, V_{DDEXT} is at its maximum, and the soft start feature is recommended to reduce the inrush currents and to reduce V_{DDINT} voltage overshoot. The approximate ramp time of V_{DDINT} will be 100 µs for each 1nF of capacitance connected to the pin in the mode. In other words, the delay in microseconds = 100 * number of nanofarads. This internal regulator mode is detailed in Figure 18-3 on page 18-18.

For VRSEL = Logic 1, External Regulator, PG Mode

When using an external regulator, the pin becomes the $\overline{\text{PG}}$ (power good, low true) signal that allows the processor to start only after the internal voltage has reached a chosen level. In this way, the startup time of the external regulator will be detected after hibernation.

If the processor never will enter the hibernate state, the pin can be grounded in this mode. This will always indicate 'power good', meaning that V_{DDINT} is at a safe operating level. Any delay required at initial power-on, to guarantee a safe operating level for V_{DDINT}, will be provided by the RESET signal.

If the external regulator for V_{DDINT} has a Power Good signal output, it can be used to help the processor recover properly from it's hibernate state. This signal may need to be inverted, as the processor's input should be low-true in order to indicate a "power good" condition.

If the external regulator does not have a Power Good output, the $\overline{\text{PG}}$ signal should be driven to a fixed level (just below the desired operating voltage) so that the $\overline{\text{PG}}$ pin voltage can be compared to V_{DDINT} by the internal startup logic. This can be accomplished with an external resistor divider from V_{DDEXT} or any other fixed stable voltage. A divider with impedance of 1M Ohm is sufficient to supply current to this $\overline{\text{PG}}$ input. To save even more current during hibernation, the VROUT signal may be used as the voltage source to the divider. With VRSEL = 1, VROUT will be low during hibernation, but will go high before the V_{DDINT} voltage is applied by the external regulator. In all cases, care should be taken to account for the min and max values of V_{DDEXT} or V_{OH} for VROUT. The voltage applied to the $\overline{\text{PG}}$ pin is used as the threshold that is compared internally to the rising value of V_{DDINT} to signal the processor to start. The voltage at $\overline{\text{PG}}$ should be calculated such that the V_{DDINT} value has risen to the desired voltage range for the application.

20 NAND FLASH CONTROLLER

The ADSP-BF52x processors provide a NAND Flash controller (NFC) as part of the external bus interface. NAND Flash devices provide high-density, low-cost memory. However, NAND Flash devices also have long random access times, invalid blocks, and lower reliability over device lifetimes.

Because of these characteristics, NAND Flash is often used for read-only code storage. In this case, all processor code can be stored in NAND Flash and then transferred to a faster memory (such as SDRAM or SRAM) before execution.

Another common use of NAND Flash is for storage of multimedia files or other large data segments. In this case, a software file system may be used to manage reading and writing of the NAND Flash device. The file system selects memory segments for storage with the goal of avoiding bad blocks and equally distributing memory accesses across all address locations.

Overview

The NFC on the ADSP-BF52x processors provides the hardware support for the combination of hardware and software necessary to interface the ADSP-BF52x processors with NAND Flash devices. The NFC provides device access timing control and hardware error checking. Software must provide bad block management, wear-leveling functions, and error correction. (See “NFC Error Detection” on page 20-9 for details on error correction.)

Bad block management includes both initial bad block detection and acquired bad block mapping. NAND Flash devices contain bad blocks that are marked by the manufacturer. Software should read the bad block information, create a table of bad block locations, and prevent use of the bad blocks. As additional blocks corrupt over time, they can be detected by the hardware and added to the bad block table by software.

When NAND Flash is used for read-write data storage, software wear-leveling is required. Wear-leveling increases the life span of NAND Flash by generating an evenly distributed number of program and erase operations across the entire memory space. Software can do this by translating logical addresses into different physical addresses for each write.

Features

Hardware features of the NFC include:

- Support for page program, page read, and block erase of NAND Flash devices, with accesses aligned to page boundaries.
- Error checking and correction (ECC) hardware that facilitates error detection and correction.
- An 8-bit external bus interface for commands, addresses and data.
- Support for SLC (single level cell) NAND Flash devices unlimited in size, with page sizes of 256 and 512 bytes. Larger page sizes can be supported in software.
- Capability of releasing external bus interface pins during long accesses.
- DMA interface to transfer data between internal memory and NAND Flash device.

Interface Overview

Figure 20-1 shows the NFC interface.

TBD

Figure 20-1. NFC Interface

The port pins used for NFC are shown in Table 20-1.

Table 20-1. NFC External Interface

Signal Name	Function	Default	Direction
D7–0	Data and Commands Bus	low	I/O
ND_CLE	Command Latch Enable	low	O
ND_ALE	Address Latch Enable	low	O
$\overline{\text{ND\_RE}}$	Read Enable	high	O
$\overline{\text{ND\_WE}}$	Write Enable	high	O
$\overline{\text{ND\_CE}}$	Chip Enable	high	O

Description of Operation

Internal Bus Interfaces

The NFC interfaces to both the peripheral bus and DAB buses on ADSP-BF52x processors.

Page reads and page writes occur over DAB. The DAB interface consists of two separate 4 word FIFOs, one for page reads and one for page writes. Each FIFO is 32-bits wide in 32-bit DMA mode. Page reads and page writes cannot be triggered at the same time. A subsequent page read or write must not be triggered until all transfers and error correction for the current page are complete.

All other accesses occur over the peripheral bus. Peripheral bus accesses always go through the NFC write buffer. This buffer is 8-bits wide. In both modes, it is 4 words deep. Software must prevent overflow of the buffer. Write buffer entries are not removed until the access is completed on the external interface. In the case of a read data request, the entry is not removed until the returned data is read from the NFC_READ register. After the fourth write to the write buffer, software must poll WB_FULL or WB_EMPTY in NFC_STAT to determine when there is additional space in the write buffer or use the WB_EMPTY interrupt to detect when the write buffer has emptied.

After reset, the peripheral bus write buffer has priority over the DAB FIFOs for access to the NFC external interface. If a page access is initiated while there are transfers in the write buffer, the page access does not start until the write buffer is empty. Likewise, once a page access starts, transfers in the write buffer do not begin until the page access is complete.

Bus Access Types

The NFC supports 8-bit NAND Flash devices. Peripheral bus accesses cause only one transfer per bus access. For DAB access, the NFC automatically breaks up 16-bit DAB transfers into multiple NAND Flash access cycles. The following table describes all the valid access types for 8-bit devices as well as the number of NAND Flash accesses it takes to complete the transaction.

Table 20-2. NFC Accesses

Bus	Bus Width	NAND Flash Width	NAND Flash Access Cycles Required
Peripheral bus	16-bit	8-bit	1
DAB	16-bit	8bit	2

Access Timing

Setup and hold times for NAND Flash accesses are described in the following table. For all signals other than D7-0, the same timing applies to $\overline{\text{ND\_WE}}$ and $\overline{\text{ND\_RE}}$.

Table 20-3. NFC Access Timing

Signal	Timing	NFC Delay Cycles	Delay at 133 MHz SCLK	Typical Requirement
$\overline{\text{ND\_CE}}$	Setup to $\overline{\text{ND\_WE}}$ falling	1 SCLK	7.5 ns	0 ns
ND_CLE, ND_ALE	Setup to $\overline{\text{ND\_WE}}$ falling	0.5 SCLK	3.75 ns	0 ns
$\overline{\text{ND\_CE}}$	Hold from $\overline{\text{ND\_WE}}$ rising	3 SCLK	22.5 ns	10 ns
ND_CLE, ND_ALE	Hold from $\overline{\text{ND\_WE}}$ rising	2.5 SCLK	18.75 ns	10 ns
D7–0	Setup to $\overline{\text{ND\_WE}}$ rising	0.5 SCLK + WR_DLY	configurable	35 ns
D7–0	Hold from $\overline{\text{ND\_WE}}$ rising	2.5 SCLK	18.75 ns	10 ns

The setup time for write data is configurable by changing WR_DLY in the NFC_CTL register. The write enable pulse width (t_{WP}) is the WR_DLY + 1 SCLK. The WR_DLY selection should be configured such that:

$t_{WP} >= \text{Max}\ (t_{WPmin}\ ,\ (t_{CS} - 1\ \text{SCLK}))$

where t_{WP} is the time for which $\overline{\text{ND\_WE}}$ is driven low, t_{WPmin} is the minimum write pulse duration from the NAND Flash data sheet, and t_{CS} is the chip enable setup time from the NAND Flash data sheet.

Likewise, the setup time for read data is configurable by changing RD_DLY in the NFC_CTL register. The read enable pulse width (t_{RP}) is the RD_DLY + 1 SCLK. The RD_DLY selection should be configured such that:

$t_{RP} > \text{Max}\ (\ t_{RPmin}\ ,\ t_{REAmax}\ ,\ (\ t_{CEAmax} - 1\ \text{SCLK})\ )$

where t_{RP} is the time for which $\overline{\texttt{ND\_RE}}$ is driven low, t_{RPmin} is the minimum read pulse duration from the NAND Flash data sheet, t_{REAmax} is the maximum read enable access time from the NAND Flash data sheet, and t_{CEAmax} is the maximum chip enable access time from the NAND Flash data sheet.

Functional Description

The following sections describe the function of the NAND Flash controller. NFC operation include:

- "Page Write" on page 20-6
- "Page Read" on page 20-7
- "Additional Operations" on page 20-8
- "Write Protection" on page 20-9
- "Chip Enable Don't Care" on page 20-9
- "NFC Error Detection" on page 20-9
- "NFC SmartMedia Support" on page 20-12

Page Write

To store data in NAND Flash, first write the program command to the NFC_CMD register. Then, write a sequence of address bits to the NFC_ADDR register. For example, an 8M x 8-bit NAND Flash device requires 22 address bits. In this case, address bits [7:0] are written, then, address bits [16:9] are written, and, finally, address bits [22:17] are written. Note that address bit [8] is generated automatically by the NAND Flash device used in this example. The automatic generation of address bit 8 is generally the case with all small page NAND flash devices. Large page NAND flash

devices however, require a different address insertion scheme. Details of the address insertion scheme are listed in the datasheet for the device. Next, set the page write start bit in the NFC_PGCTL register. This initiates DMA transfers to complete the page write. After writing all of the data, software can append the ECC values from the ECC registers to store them in the spare area of the NAND Flash. Finally, the page program confirm command is written to NFC_CMD to initiate the NAND Flash programming process. The NAND Flash asserts $\overline{\text{ND\_RB}}$ until the page is completely programmed. At that time, the Write Status Bit in the NAND Flash device may be checked. The following figure shows the timing of a NAND Flash write access.

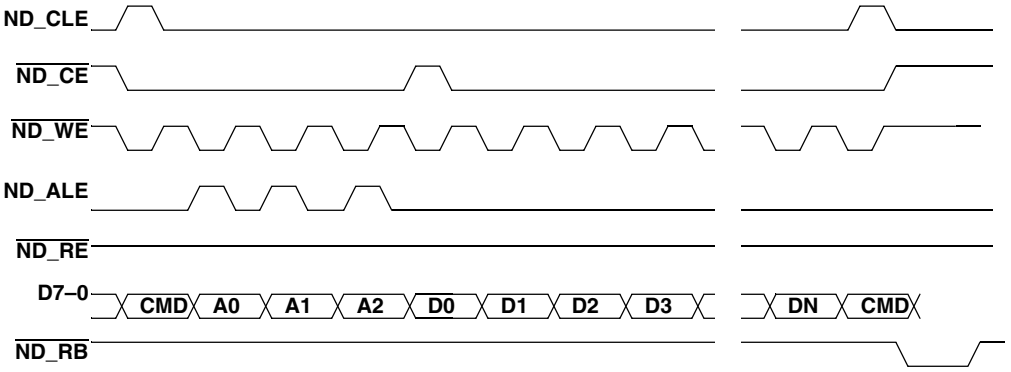


Figure 20-2. NAND Flash Program Operation

Page Read

To read data from NAND Flash, first write the read command to the NFC_CMD register. Then write a sequence of address bits to the NFC_ADDR register. For an 8M x 8-bit example, write address bits [7:0], address bits [15:8], and address bits [21:16]. Next, wait for a rising edge of $\overline{\text{ND\_RB}}$, indicating that the requested data is available. Then, set the page read start bit in NFC_PGCTL. This initiates the DMA transfers for a page read. As each read occurs, new ECC values are calculated for each 256 or 512 byte page.

When the page read is complete, the core must complete final data read requests to obtain the stored ECC values which were written in the spare area when the page was programmed. Software can compare this to the new ECC values to determine if any bit errors have occurred. The following figure shows the timing of a NAND Flash read access.

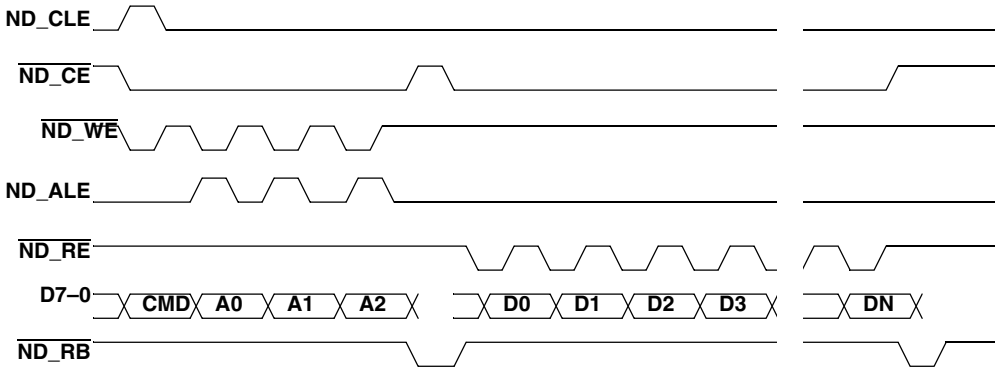


Figure 20-3. NAND Flash Read Operation

Additional Operations

The core may execute block erase, read status, read ID, and reset directly, without using DMA. Commands must be written to NFC_CMD, addresses must be written to NFC_ADDR, and data must be written to NFC_DATA_WR. Data reads are requested by first writing to NFC_DATA_RD and then reading back from NFC_READ.

To check that an operation is complete, $\overline{\text{ND\_RB}}$ may be polled in the NFC_STAT register or used to trigger an interrupt. The NFC samples $\overline{\text{ND\_RB}}$ before beginning a page read or page write sequence. At all other times, software must sample $\overline{\text{ND\_RB}}$ before performing an access.

See NAND Flash device data sheets for examples of these operations.

Write Protection

NAND Flash devices require a write protection input signal (nWP) to prevent inadvertent write or erase operations. A GPIO can be used for this purpose.

Chip Enable Don't Care

Some NAND Flash devices ignore the read enable, write enable, command latch enable, and address latch enable control signals when chip select is de-asserted during page reads and page programs. These devices are called chip enable don't care (CEDC) NAND Flash devices. This is the only type of device supported by the NFC.

NFC Error Detection

The NFC error checking and correction (ECC) logic can detect one bit of correctable error or multiple bits of non-correctable error. The NFC employs a Hamming code algorithm, which generates two sets of parity bits for every 256 bytes of data. For 512 byte pages, the page is split into two halves, and separate ECC values are calculated for each half.

For every 256 bytes of data, 22 bits of ECC parity data are generated as follows:

```
P1 = D[1] ^ D[3] ^ D[5] ^ D[7] ^ D[9] …^ D[2047];
P2 = D[2] ^ D[3] ^ D[6] ^ D[7] ^ D[10] ^ D[11] … ^ D[2042] ^
D[2043] ^ D[2046] ^ D[2047];
P4 = D[4] ^ D[5] ^ D[6] ^ D[7] ^ D[12] ^ D[13] ^ D[14] ^ D[15] ^
D[20] ^ D[21] ^ D[22] ^ D[23] … ^ D[2044] ^ D[2045] ^ D[2046] ^
D[2047];
P8 = D[8] ^ D[9] ^ D[10] ^ D[11] ^ D[12] ^ D[13] ^ D[14] ^ D[15]
^ D[24] ^ D[25] ^ D[26] D[27] ^ D[28] ^ D[29] ^ D[30] ^ D[31] … ^
D[2040] ^ D[2041] ^ D[2042] ^ D[2043] ^ D[2044] ^ D[2045] ^
D[2046] ^ D[2047];
```

```
…
…
P1’ = D[0] ^ D[2] ^ D[4] ^ D[6] ^ D[8] … ^ D[2046];
P2’ = D[0] ^ D[1] ^ D[4] ^ D[5] ^ D[8] ^ D[9] … ^ D[2040] ^
D[2041] ^ D[2044] ^ D[2045];
P4’ = D[0] ^ D[1] ^ D[2] ^ D[3] ^ D[8] ^ D[9] ^ D[10] ^ D[11] ^
D[16] ^ D[17] ^ D[18] ^ D[19] … ^ D[2040] ^ D[2041] ^ D[2042] ^
D[2043];
…
…
```

In this way, P1, P2, P4, P8, P16, P32, P64, P128, P256, P512, and P1024 as well as P1’, P2’, P4’, P8’, P16’, P32’, P64’, P128’, P256’, P512’, and P1024’ are calculated, producing a total of 22 parity bits for each 256 bytes (2048 bits) of data.

The NFC writes this 22-bit ECC value into the `NFC_ECCx` registers. Software can store these values in the spare area of the NAND Flash device for later comparison. When reading back data, the NFC automatically calculates new ECC values from the received data. Software can generate error syndromes by exclusive OR’ing the stored and newly calculated ECC values.

Error Analysis

Analyzing the ECC values lets you determine the error syndrome. The resulting error syndromes indicate what type of data errors have occurred.

For example, when a 256 byte page is read back, ECC0(stored) contains the parity bits stored read from the spare area. ECC1(stored) contains the parity’ bits read from the spare area. Similarly, ECC0(calculated) and ECC1(calculated) contain the newly calculated parity and parity’ bits,

respectively. To interpret the ECC values, software generates the following error syndromes:

```
syndrome0[21:0] = {ECC0calculated[10:0],ECC1calculated[10:0]} ^
{ECC0stored[10:0],ECC1stored[10:0]}
syndrome1[10:0] = ECC0calculated[10:0] ^ ECC0stored[10:0]
syndrome2[10:0] = ECC0calculated[10:0] ^ ECC1calculated[10:0]
syndrome3[10:0] = ECC0stored[10:0] ^ ECC1stored[10:0]
syndrome4[10:0] = syndrome2[10:0] ^ syndrome3[10:0]
```

Syndrome 0 indicates whether there is an error in the data. Syndrome 4 indicates whether the error is a one-bit correctable error. Syndrome 1 indicates the bit location of any one-bit errors. After calculating these syndromes, software must examine their values and take the appropriate actions.

- If Syndrome 0 is 0x000, the data is valid and no actions are required.
- If Syndrome 0 has exactly 11 bits that are one and Syndrome 4 is 0x7FF, there is a one bit correctable error. Syndrome 1 gives the failing bit number. For example, if Syndrome 1 is 46, the seventh bit in the sixth byte transferred needs to be inverted.
- If Syndrome 0 has only one bit that is one, there is an error in the ECC data itself. No action is required, since ECC data is discarded after each page read, but no error checking can be done.
- If Syndrome 0 has any other value, there is a multiple-bit, unrecoverable error. Software should mark the block containing this page as a bad block.

Examples of possible Syndrome 0 values are shown in the following table.

Table 20-4. ECC Syndrome Examples

Syndrome 0	Type of Value	Meaning	Action Required
0x000000	All zero	No Error in Data	None
0x2CCA66	Exactly 11 bits are one, each parity and parity' pair is 1 & 0 or 0 & 1	1-bit Correctable Error	Correct Error
0x000040	Only one bit is one	ECC Data was incorrect	None
0x06B35A	Random data	More than 1-bit Error, non-correctable error	Discard Data, Mark Bad Block

Large Page Size Support

Page sizes larger than 512 bytes can be supported by NFC as long as they require only 1-bit error correction per 512 bytes. For example, a 2K byte page can be accessed by treating it as four 512-byte pages. The page program and page reads must be conducted as four 512-byte accesses, and the ECC values for each 512 bytes of data must be read back from the ECC registers and then temporarily stored. The ECC registers must be reset before the next 512 bytes are transferred. Once ECC values from all four 512-byte pages are calculated, they must be written into the NAND Flash spare area for a page read or compared to those in the spare area for a page program.

NFC SmartMedia Support

NAND Flash and SmartMedia devices have nearly identical interfaces. The main difference is that SmartMedia devices are removable, and, therefore, require card insertion, card ejection and write protection signals. On ADSP-BF52x processors, these features can be supported using GPIOs.

Programming Model

The following sections describe the NAND Flash controller's programming model.

Before using the NFC, pins with GPIO functions must be configured to select the NFC functionality. This causes a rising edge detect on $\overline{\text{ND\_RB}}$, which must be cleared before beginning NFC programming sequences.

To conduct a page read, the core may use the following procedure:

1. The core writes to the appropriate DMA registers to enable the NFC DMA channel for receive mode and to configure the correct number of transfers for a single page.
2. The core sets up the appropriate configuration by writing the NFC_CTL register.
3. The core clears the NFC_ECCx registers by setting the ECC_RST bit in the NFC_RST register.
4. The core writes the page read commands to NFC_CMD register and the page addresses to the NFC_ADDR register (maximum of four writes at a time).
5. The core waits for a rising edge detection on $\overline{\text{ND\_RB}}$.
6. The core sets the page read start bit in the NFC_PGCTL register.
7. When the DMA generates an interrupt on completion, the core reads the remaining spare bytes.
8. The core compares ECC information stored in the spare bytes to the ECC register values calculated during the page read.
9. If there is an ECC error, the core must correct the corrupted data.

To conduct a page write, the core may use the following procedure:

1. The core writes to the appropriate DMA registers to enable the NFC DMA channel for transmit mode and to configure the correct number of transfers for a single page.
2. The core sets up the appropriate configuration by writing the NFC_CTL register.
3. The core clears the NFC_ECCx registers by setting the ECC_RST bit in the NFC_RST register.
4. The core writes the page write commands to NFC_CMD register and the page addresses to the NFC_ADDR register (maximum of 4 writes at a time).
5. The core waits for the write buffer to be empty by either polling the status bit or waiting for the WB_EDGE interrupt.
6. The core sets the page write start bit in the NFC_PGCTL register.
7. When the DMA generates an interrupt on completion, the WR_DONE bit should be checked to verify the last transfer is complete, then the core reads the ECC register values and writes those values to the spare bytes of the page.
8. The core writes the page program confirm command to the NFC_CMD register.
9. The core waits for the write buffer to empty and for a subsequent rising edge detection on $\overline{\text{ND\_RB}}$.

NFC Registers

The NFC has a group of memory-mapped registers (MMRs) that regulate its operation. These registers are listed in Table 20-5.

Descriptions and bit diagrams for each of these MMRs are provided in the following sections. The NFC MMRs start at a base address of 0xFFC0 3700.

The NFC_ECCx and NFC_COUNT registers should not be read while an access to NAND Flash is happening on the EBIU. Otherwise, the registers may be updating during a read and coherency of the register bits is not guaranteed.

Table 20-5 lists all of the NFC memory mapped registers.

Table 20-5. NFC Memory Mapped Registers

Register Name	Address	Description
NFC_CTL	0xFFC0 3700	Control Register
NFC_STAT	0xFFC0 3704	Status Register
NFC_IRQSTAT	0xFFC0 3708	Interrupt Status Register
NFC_IRQMASK	0xFFC0 370C	Interrupt Mask Register
NFC_ECC0	0xFFC0 3710	ECC Register 0
NFC_ECC1	0xFFC0 3714	ECC Register 1
NFC_ECC2	0xFFC0 3718	ECC Register 2
NFC_ECC3	0xFFC0 371C	ECC Register 3
NFC_COUNT	0xFFC0 3720	ECC Count Register
NFC_RST	0xFFC0 3724	ECC Reset Register
NFC_PGCTL	0xFFC0 3728	Page Control Register
NFC_READ	0xFCC0 372C	Read Data Register
NFC_ADDR	0xFFC0 3740	Address Register
NFC_CMD	0xFFC0 3744	Command Register
NFC_DATA_WR	0xFFC0 3748	Data Write Register
NFC_DATA_RD	0xFFC0 374C	Data Read Register

NFC Control (NFC_CTL) Register

The NFC_CTL register (see Figure 20-4) contains timing and mode configuration fields. The read strobe delay (RD_DLY) and write strobe delay (WR_DLY) fields extend the $\overline{\text{ND\_RE}}$ and $\overline{\text{ND\_WE}}$ strobes, respectively, by the specified number of cycles. If no extension is specified, $\overline{\text{ND\_RE}}$ and $\overline{\text{ND\_WE}}$ assert for a single SCLK cycle. The NAND data width (NWIDTH) bit selects the data bus width size of the external NAND Flash device. The page size (PG_SIZE) bit determines where the ECC data values are written. For a 256-byte page, ECC values are always calculated in NFC_ECC0 and NFC_ECC1. For a 512-byte page, the first ECC value is calculated in NFC_ECC0 and NFC_ECC1 while the next ECC value is calculated in NFC_ECC2 and NFC_ECC3.

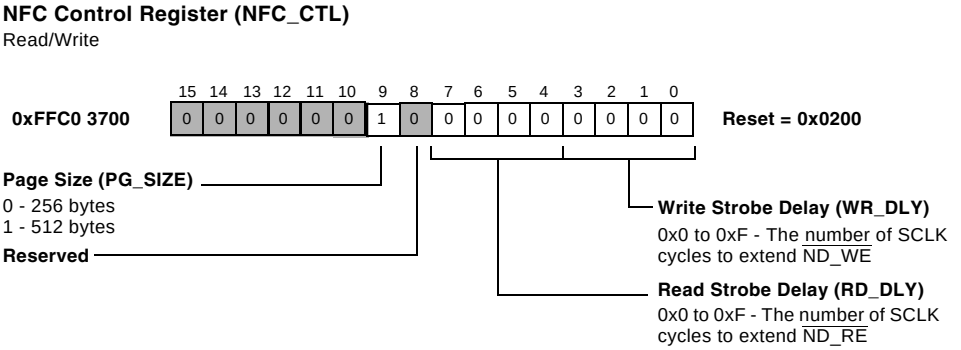

NFC Control Register (NFC_CTL)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 3700
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
Reset = 0x0200
Page Size (PG_SIZE)
0 - 256 bytes
1 - 512 bytes
Reserved
Write Strobe Delay (WR_DLY)
0x0 to 0xF - The number of SCLK cycles to extend ND_WE
Read Strobe Delay (RD_DLY)
0x0 to 0xF - The number of SCLK cycles to extend ND_RE

Figure 20-4. NFC Control Register

NFC Status (NFC_STAT) Register

The NFC_STAT register (see Figure 20-5) contains status information. The NBUSY bit contains the synchronized value of the $\overline{\text{ND\_RB}}$ pin. The write buffer empty (WB_EMPTY) and write buffer full (WB_FULL) status bits contain write buffer status information. When WB_FULL is set, writes to any write buffer register are ignored and cause the WB_OVF bit in the NFC_IRQSTAT register to be set. The page write pending (PG_WR_STAT) and page read

pending (PG_RD_STAT) bits show indicate that a page write (or read) has been started and not completed.

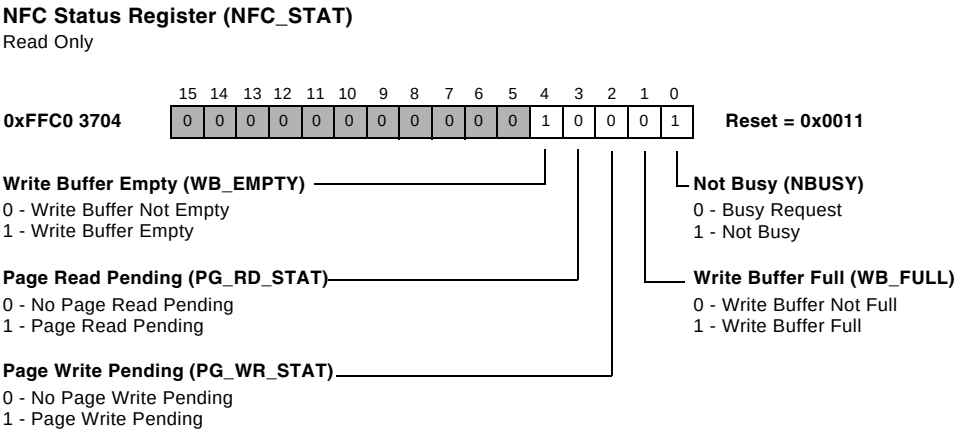

NFC Status Register (NFC_STAT)
Read Only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 3704
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1
Reset = 0x0011
Write Buffer Empty (WB_EMPTY)
0 - Write Buffer Not Empty
1 - Write Buffer Empty
Page Read Pending (PG_RD_STAT)
0 - No Page Read Pending
1 - Page Read Pending
Page Write Pending (PG_WR_STAT)
0 - No Page Write Pending
1 - Page Write Pending
Not Busy (NBUSY)
0 - Busy Request
1 - Not Busy
Write Buffer Full (WB_FULL)
0 - Write Buffer Not Full
1 - Write Buffer Full

Figure 20-5. NFC Status Register

NFC Interrupt Status (NFC_IRQSTAT) Register

The NFC_IRQSTAT register (see Figure 20-6) reports the status of additional NFC interrupt sources. All bits in this register are write-1-to-clear. The NBUSYIRQ sticky bit is asserted when a rising edge is detected on the $\overline{\text{ND\_RB}}$ signal. This bit must be cleared (W1C) before starting a new access. The WB_OVF bit is asserted when the write buffer overflows and indicates an error condition. The write buffer edge detect (WB_EDGE) is set when the write buffer transitions from not empty to empty. The read data ready (RD_RDY) bit indicates that a read data command has completed and that data is available for reading from the NFC_READ register. The page write

done (WR_DONE) bit indicates that a page write has completed and that the last access in the page has been transferred on the external bus.

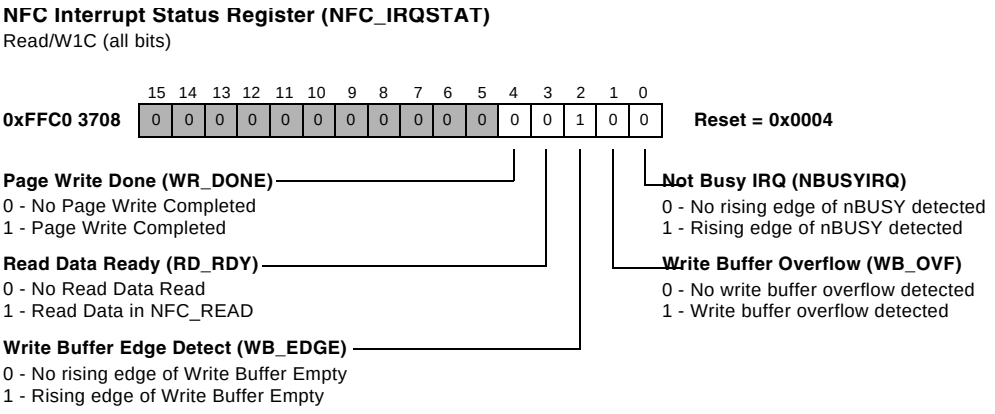

NFC Interrupt Status Register (NFC_IRQSTAT)
Read/W1C (all bits)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 3708
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
Reset = 0x0004
Page Write Done (WR_DONE)
0 - No Page Write Completed
1 - Page Write Completed
Read Data Ready (RD_RDY)
0 - No Read Data Read
1 - Read Data in NFC_READ
Write Buffer Edge Detect (WB_EDGE)
0 - No rising edge of Write Buffer Empty
1 - Rising edge of Write Buffer Empty
Not Busy IRQ (NBUSYIRQ)
0 - No rising edge of nBUSY detected
1 - Rising edge of nBUSY detected
Write Buffer Overflow (WB_OVF)
0 - No write buffer overflow detected
1 - Write buffer overflow detected

Figure 20-6. NFC Interrupt Status Register

NFC Interrupt Mask (NFC_IRQMASK) Register

The NFC_IRQMASK register (see Figure 20-7) contains individual mask bits for each NFC interrupt source. After masking, the bits are OR-ed together and routed to the system interrupt controller.

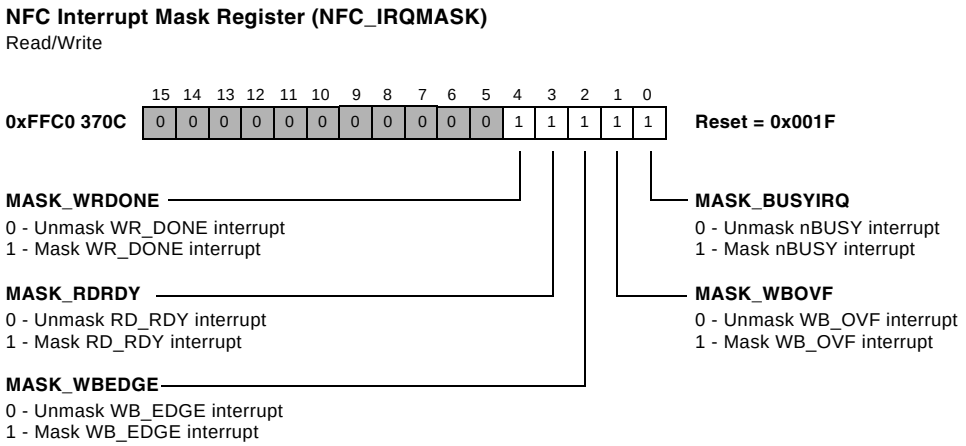

NFC Interrupt Mask Register (NFC_IRQMASK)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 370C
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
Reset = 0x001F
MASK_WRDONE
0 - Unmask WR_DONE interrupt
1 - Mask WR_DONE interrupt
MASK_RDRDY
0 - Unmask RD_RDY interrupt
1 - Mask RD_RDY interrupt
MASK_WBEDGE
0 - Unmask WB_EDGE interrupt
1 - Mask WB_EDGE interrupt
MASK_BUSYIRQ
0 - Unmask nBUSY interrupt
1 - Mask nBUSY interrupt
MASK_WBOVF
0 - Unmask WB_OVF interrupt
1 - Mask WB_OVF interrupt

Figure 20-7. NFC Interrupt Mask Register

NFC ECC (NFC_ECCx) Registers

The NFC_ECCx registers (see Figure 20-8) contain the 22-bit ECC parity values calculated for data read from or written to the NAND Flash device. When data is written, the processor must store these values in the spare area of the NAND Flash device. When data is read, the ECC values are calculated for comparison with the values stored in the spare area.

The four 16-bit ECC registers are used to hold the ECC data as it is calculated by the ECC logic. NFC_ECC0 and NFC_ECC1 are used to hold the 22-bit ECC value for the first 256 bytes of the page. For 512-byte pages, NFC_ECC2 and NFC_ECC3 hold the 22-bit ECC value for the second half-page (256 bytes). The page size is configured in NFC_CTL.

The values in the ECC registers are updated on every cycle that data is transferred. They are not updated when spare area bytes are read or written. NFC_ECC0 and NFC_ECC1 are valid after the transfer of the 256th byte in a page. NFC_ECC2 and NFC_ECC3 are valid after the transfer of the 512th byte in a page.

Note that the ECC registers are 16 bits each. When writing the ECC value to an 8-bit NAND Flash device, the lower eight bits must be written first, followed by the upper eight bits.

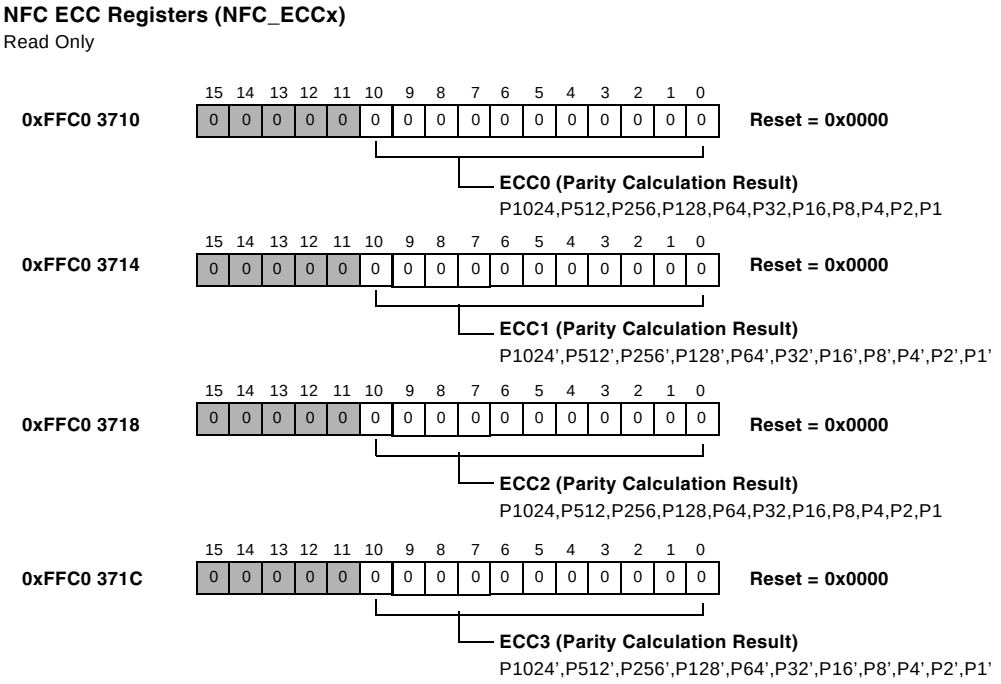


Figure 20-8. NFC ECC Registers

NFC Count (NFC_COUNT) Register

The NFC_COUNT register (see Figure 20-9) reports the number of bytes that have been transferred in the current page. The count starts at one and

increments up to 512 for a 512-byte page. This register is used primarily for debugging purposes. The counter is reset when the ECC_RST bit in the NFC_RST register is set.

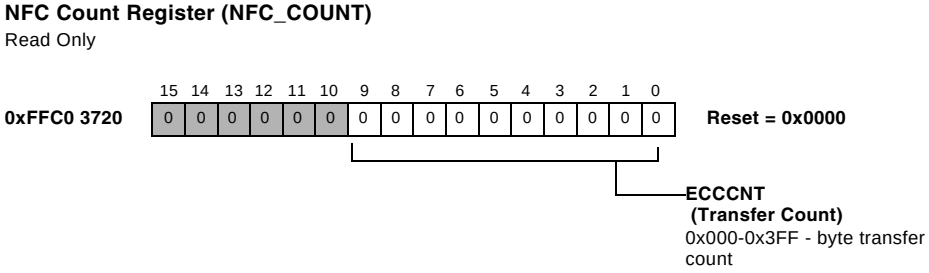

NFC Count Register (NFC_COUNT)
Read Only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 3720
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
ECCCNT
(Transfer Count)
0x000-0x3FF - byte transfer count

Figure 20-9. NFC Count Register

NFC Reset (NFC_RST) Register

The NFC_RST register (see Figure 20-10) allows software to reset the ECC registers and the NFC counters. This register must be written before each page is transferred to generate the correct ECC register values. The ECC reset bit is automatically cleared by the NFC on completion of the reset.

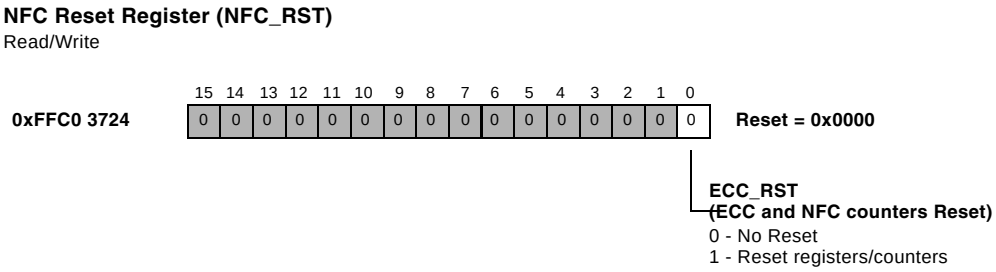

NFC Reset Register (NFC_RST)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 3724
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
ECC_RST
(ECC and NFC counters Reset)
0 - No Reset
1 - Reset registers/counters

Figure 20-10. NFC Reset Register

NFC Page Control (NFC_PGCTL) Register

The NFC_PGCTL register (see Figure 20-11) allows the processor to initiate page reads or writes. All bits in the register are write only. The page data is

always transferred using the DAB bus. When either a page read or page write is pending, page read start (PG_RD_START) and page write start (PG_WR_START) are ignored.

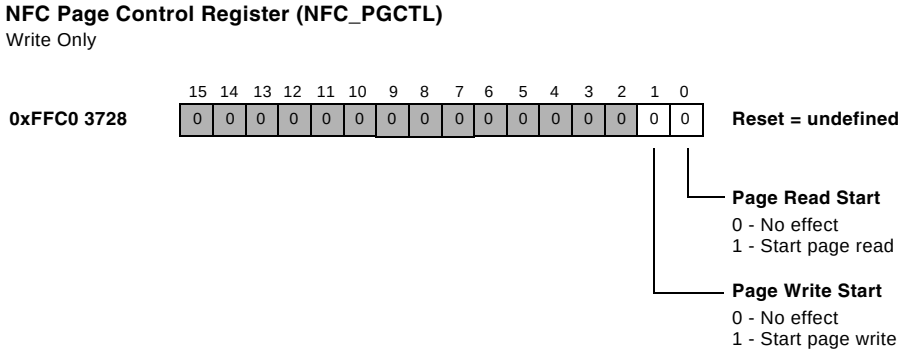

NFC Page Control Register (NFC_PGCTL)
Write Only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 3728
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = undefined
Page Read Start
0 - No effect
1 - Start page read
Page Write Start
0 - No effect
1 - Start page write

Figure 20-11. NFC Page Control Register

NFC Read Data (NFC_READ) Register

The NFC_READ register (see Figure 20-12) contains read data returned from the NAND flash after a read is requested using the NFC_DATA_RD register. Only the eight LSB have valid data. The RD_RDY status bit and interrupt indicate when new data is available for reading.

To prevent overflow of NFC_READ, the read data request is not removed from the write buffer until the returned data is read back from NFC_READ.

As a result, no other commands, address or data are sent to the NAND Flash while the read data request is active in the write buffer.

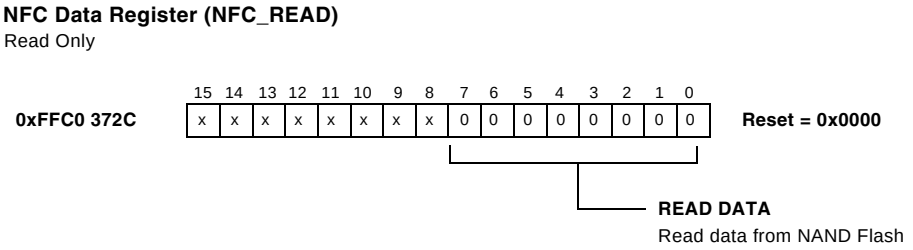

NFC Data Register (NFC_READ)
Read Only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 372C
x x x x x x x x 0 0 0 0 0 0 0 0
Reset = 0x0000
READ DATA
Read data from NAND Flash

Figure 20-12. NFC Read Data Register

NFC Address (NFC_ADDR) Register

The NFC_ADDR register (see Figure 20-13) contains address bits to be sent to the NAND flash device. Only the eight LSB of the written address are sent to the NAND flash device, the upper eight bits are ignored. Values written to this register are stored in the NFC write buffer.

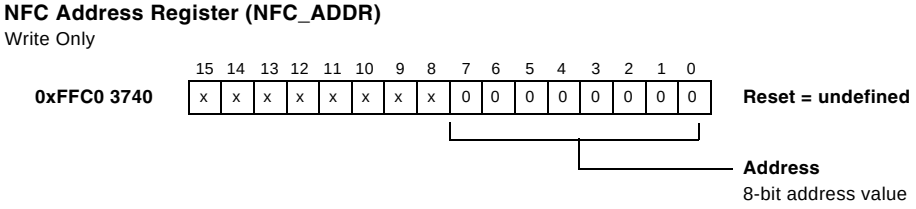

NFC Address Register (NFC_ADDR)
Write Only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 3740
x x x x x x x x 0 0 0 0 0 0 0 0
Reset = undefined
Address
8-bit address value

Figure 20-13. NFC Address Register

NFC Command (NFC_CMD) Register

The NFC_CMD register (see Figure 20-14) contains commands to be written to the NAND Flash device. Only the eight LSB of the written command

are sent to the NAND flash device, the upper eight bits are ignored. Values written to this register are stored in the NFC write buffer.

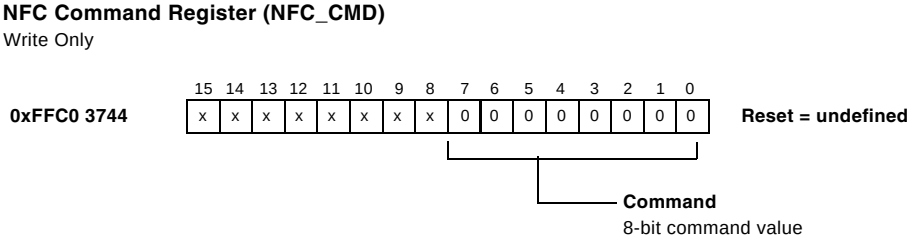

NFC Command Register (NFC_CMD)
Write Only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 3744
x x x x x x x x 0 0 0 0 0 0 0 0
Reset = undefined
Command
8-bit command value

Figure 20-14. NFC Command Register

NFC Data Write (NFC_DATA_WR) Register

The NFC_DATA_WR register (see Figure 20-15) contains data to be written to the NAND Flash device. Only the eight LSB of the written command are sent to the NAND flash device, the upper eight bits are ignored. Values written to this register are stored in the NFC write buffer.

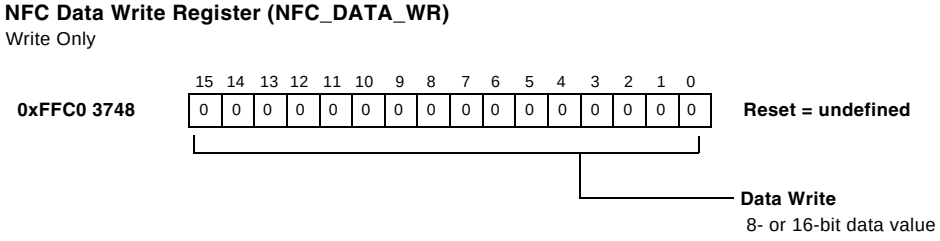

NFC Data Write Register (NFC_DATA_WR)
Write Only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 3748
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = undefined
Data Write
8- or 16-bit data value

Figure 20-15. NFC Data Write Register

NFC Data Read (NFC_DATA_RD) Register

The NFC_DATA_RD register (see Figure 20-16) triggers a read request to the NAND Flash device. The data written is ignored. The read request from this register is stored in the NFC write buffer.

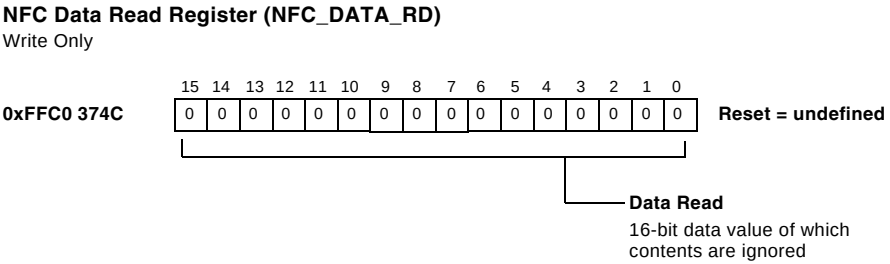

NFC Data Read Register (NFC_DATA_RD)
Write Only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 374C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = undefined
Data Read
16-bit data value of which
contents are ignored

Figure 20-16. NFC Data Read Register

NFC Programming Examples

The following examples show how to use the NFC.

Listing 20-1. Example for write:

```
//Page Write
  P0.L = lo(NFC_CTL);
  P0.H = hi(NFC_CTL);
  R0 = 0x0200(z);
  W[P0] = R0.L;
  P0.L = lo(NFC_IRQMASK);
  P0.H = hi(NFC_IRQMASK);
  R0 = 0x0000(z);
  W[P0] = R0.L;

  P0.L = lo(NFC_RST);
```

```
  P0.H = hi(NFC_RST);
  R0 = 0x1(z);
  W[P0] = R0.L;
  P0.L = lo(NFC_CMD);
  P0.H = hi(NFC_CMD);
  R0 = 0x80(z);
  W[P0] = R0.L;
  P1.L = lo(NFC_ADDR);
  P1.H = hi(NFC_ADDR);
  R1 = 0x0(z);
  W[P1] = R1.L;
  W[P1] = R1.L;
  P2.L = lo(NFC_STAT);
  P2.H = hi(NFC_STAT);
wr_buf_fullb:
  R5.L = W[P2];
  CC = BITTST(R5,1);
  IF CC JUMP wr_buf_fullb;
  W[P1] = R1.L;
  W[P1] = R1.L;
  W[P1] = R1.L;
  p5.l = 0xffff;
  p5.h = 0xffff;
  lsetup(delay3, delay3) lc0 = p5;
  delay3:  nop;
  P2.L = lo(NFC_PGCTL);
  P2.H = hi(NFC_PGCTL);
  R2.L = W[P2];
  R2 = 0x0002(z);
  W[P2] = R2.L;
```

Listing 20-2. Example for read:

```
//Page Read
```

```
    P0.L = lo(NFC_CTL);
    P0.H = hi(NFC_CTL);
    R0 = 0x0250(z);
    W[P0] = R0.L;

    P0.L = lo(NFC_IRQMASK);
    P0.H = hi(NFC_IRQMASK);
    R0 = 0x0000(z);
    W[P0] = R0.L;

    P0.L = lo(NFC_RST);
    P0.H = hi(NFC_RST);
    R0 = 0x1(z);
    W[P0] = R0;

    P0.L = lo(NFC_CMD);
    P0.H = hi(NFC_CMD);
    R0 = 0x0(z);
    W[P0] = R0;

    P1.L = lo(NFC_ADDR);
    P1.H = hi(NFC_ADDR);
    R1 = 0x0(z);
    W[P1] = R1;
    W[P1] = R1;

  wr_buf_full2:
    P2.L = lo(NFC_STAT);
    P2.H = hi(NFC_STAT);
    R5.L = W[P2];
    CC = BITTST(R5,0);
    IF !CC JUMP wr_buf_full2;

    W[P1] = R1;
```

```
  W[P1] = R1;
  W[P1] = R1;

wr_buf_full5:
  P2.L = lo(NFC_STAT);
  P2.H = hi(NFC_STAT);
  R5.L = W[P2];
  CC = BITTST(R5,0);
  IF !CC JUMP wr_buf_full5;

  P0.L = lo(NFC_CMD);
  P0.H = hi(NFC_CMD);
  R0 = 0x30(z);
  W[P0] = R0;

  p5.l = 0xffff;
  p5.h = 0x0000;
  lsetup(st, st) lc0 = p5;
  st:  nop;

  P2.L = lo(NFC_PGCTL);
  P2.H = hi(NFC_PGCTL);
  R1 = 0x1(z);
  W[P2] = R1.L;
```

21 ETHERNET MAC

This chapter describes the Ethernet Media Access Controller (EMAC) peripheral. Following an overview and list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF52x

For details regarding the EMAC for the ADSP-BF52x product, please refer to the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet*.

For EMAC DMA channel assignments, refer to Table 6-7 on page 6-106 in Chapter 6, “Direct Memory Access”.

For EMAC interrupt vector assignments, refer to Table 5-3 on page 5-18 in Chapter 5, “System Interrupts”.

To determine how the EMAC is multiplexed with other functional pins, refer to Table 9-2 on page 9-5 through Table 9-5 on page 9-9 in Chapter 9, “General-Purpose Ports”.

For a list of MMR addresses for the EMAC, refer to Appendix A, “System MMR Assignments”.

EMAC behavior for the ADSP-BF52x that differs from the general information in this chapter can be found at the end of this chapter in the

section “Unique Behavior for the ADSP-BF52x Processor” on page 21-128.

Overview

The Ethernet MAC provides a 10/100M bit/s Ethernet interface, compliant to IEEE Std. 802.3-2002, between an MII (Media Independent Interface) and the Blackfin peripheral subsystem.

Features

The Ethernet MAC includes these features:

- Independent DMA-driven RX and TX channels
- MII/RMII interface
- 10M bit/s and 100M bit/s operation (full or half duplex)
- VLAN support (full or half duplex)
- Automatic network monitoring statistics
- Flexible address filtering
- Flexible event detection for interrupt handling
- Validation of IP and TCP (payload) checksum
- Remote-wakeup Ethernet frames
- Network-aware system power management

The MAC is fully compliant to IEEE Std. 802.3-2002.

Interface Overview

Figure 21-1 illustrates the overall architecture of the Ethernet controller. The central MAC block implements the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol for both half-duplex and full-duplex modes.

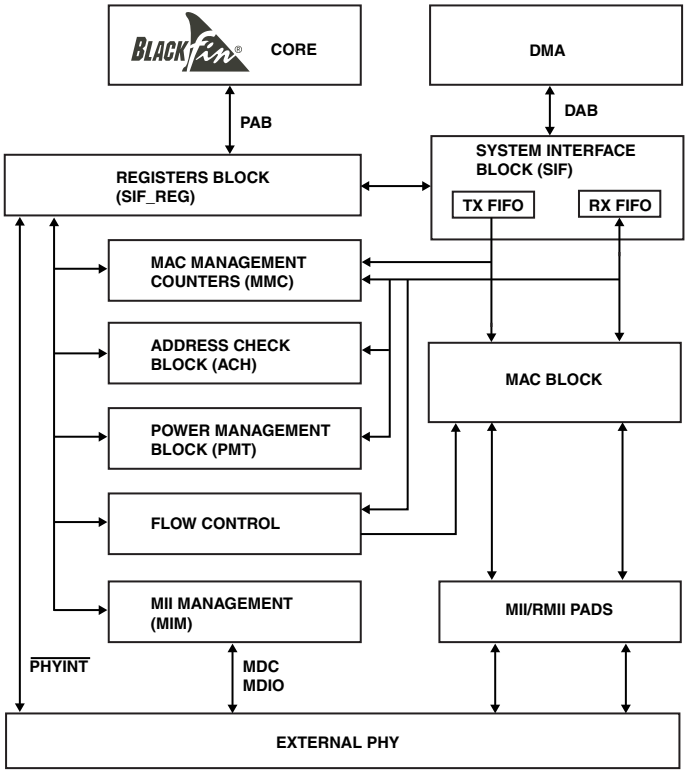


Figure 21-1. Ethernet MAC Block Diagram

The System Interface (SIF) block contains FIFOs for RX and TX data and handles the synchronization of data between the MAC RX and TX data streams and the Blackfin DMA controller.

The System Interface Registers (SIF_REG) block is an interface from the Blackfin peripheral access bus to the internal registers in the MAC. This block also generates the Ethernet event interrupt, and supports the PHYINT pin by which the PHY can notify the Blackfin processor when the PHY detects changes to the link status, such as auto-negotiation or duplex mode change.

The MAC Management Counters (MMC) block is an extended set of registers that collect various statistics compliant with IEEE 802.3 definitions regarding the operation of the interface. They are updated for each new transmitted or received frame.

The Power Management (PMT) block adds support for wakeup frames and magic packet technology that allows waking up the processor from low power operating modes. Further details regarding these low-power operating modes and voltage regulator wakeup functionality can be found in Chapter 18, "Dynamic Power Management".

The Address Check (ACH) block checks the destination address field of all incoming packets. Based on the type of address filtering selected, this indicates the result of the address checking to the MAC block.

The MII Management (MIM) block handles all transactions to the control and status registers on the external PHY.

External Interface

The following sections describe the external interface.

Clocking

The Ethernet MAC is clocked internally from SCLK on the processor. A buffered version of CLKIN may be used to drive the external PHY via the CLKBUF pin. See Figure 21-2.

The CLKBUF signal is not generated by a PLL and supports jitter and stability functions comparable to XTAL. The CLKBUF pin is enabled by the CLKBUFOE bit in the VR_CTL register. See Chapter 18, “Dynamic Power Management” for more information.

A 25 MHz clock (whether driven with the CLKBUF pin or an external crystal) should be used with an MII PHY. A 50 MHz clock source is required to drive an RMII PHY.

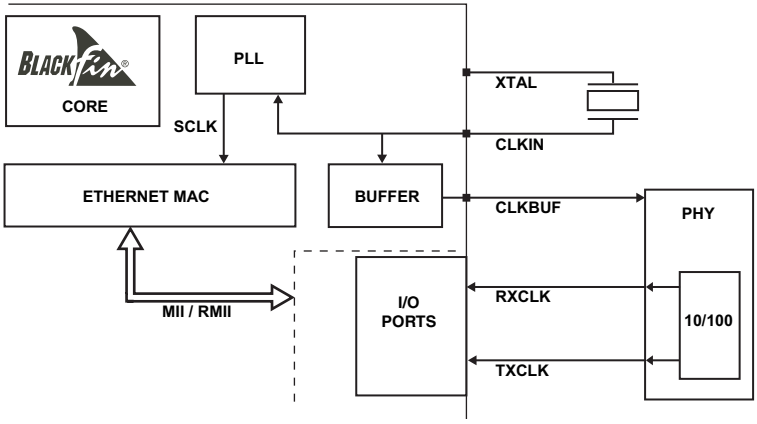


Figure 21-2. Clock Function Diagram

Pins

MII and RMII peripherals are multiplexed into the general-purpose ports. To use MII and RMII operations, set the appropriate I/O Port registers accordingly. See Chapter 9, “General-Purpose Ports” for more information.

IEEE802.3-2002, section two, clause 22.2.1.6, characterizes the MII TX_ER pin as an option useful only for certain applications (for example, repeater applications). Therefore, the TX_ER pin is not present in this design.

Table 21-1 shows the pins for the MAC.

Table 21-1. Ethernet MAC Pins

MII Signal Name	MII Input/ Output	RMII Multiplexed Name	RMII Input/ Output	Description
MII CRS	I	RMII CRS_DV	I	Ethernet MII carrier sense/RMII carrier sense and receive data valid
MII RXER	I	RMII RXER	I	Ethernet MII or RMII receive error
MDIO	I/O	MDIO	I/O	Ethernet management channel serial data
MII TXEN	O	RMII TXEN	O	Ethernet MII or RMII transmit enable
MII TXCLK	I	RMII REFCLK	I	Ethernet MII transmit clock/RMII reference clock
MII TXD0	O	RMII TXD0	O	Ethernet MII or RMII transmit D0
MII RXD0	I	RMII RXD0	I	Ethernet MII or RMII receive D0
MII TXD1	O	RMII TXD1	O	Ethernet MII or RMII transmit D1
MII RXD1	I	RMII RXD1	I	Ethernet MII or RMII receive D1
MII TXD2	O			Ethernet MII transmit D2
MII RXD2	I			Ethernet MII receive D2
MII TXD3	O			Ethernet MII transmit D3
MII RXD3	I			Ethernet MII receive D3
MII RXCLK	I			Ethernet MII receive clock
MII RXDV	I			Ethernet MII receive data valid
MII COL	I			Ethernet collision
MDC	O	MDC	O	Ethernet management channel clock
MII PHY-INT	I/O	RMII MDINT	I	Ethernet MII PHY interrupt/RMII management data interrupt

Internal Interface

Communication between the MAC and the Blackfin processor peripheral subsystem takes place over the peripheral bus and the DMA Access Bus (DAB). The peripheral bus is used by the Blackfin processor core to configure and monitor the peripheral's control and status registers. All data transfers to and from the peripheral are handled by the Blackfin DMA controller and take place via the DAB.

Power Management

The processor provides power management states which allow programming the MAC to wake the processor upon reception of specific Ethernet frames and/or upon selected events detected by the PHY. The MAC itself requires no additional power management intervention; its internal clocks power down automatically when not required. The MAC clocks run in any of these conditions (provided the processor is in the sleep, active, or full on state):

1. Either the receiver or transmitter is enabled (RE or TE = 1)
2. During an MII Management transfer (on MDC/MDIO)
3. During a core access to an MAC control/status register
4. While PHY interrupts are enabled in the MAC (PHYIE in the EMAC_SYSCTL register is set)

Description of Operation

The following sections describe the operation of the MAC.

Protocol

The Ethernet MAC complies with IEEE Std. 802.3-2002. The MII management interface is described below.

MII Management Interface

The IEEE 802.3 MII management interface, also known as the MDIO station management interface, allows the Blackfin processor to monitor and control one or more external Ethernet physical-layer transceivers (PHYs). The MII management interface physically consists of a 2-wire serial connection composed of the MDC (management data clock) output signal and the MDIO (management data input/output) bidirectional data signal. See Figure 21-3 and Figure 21-4.

The MII management logical interface specifies:

- A set of 16-bit device control/status registers within PHYs, including both required registers with standardized bit definitions as well as optional vendor-specified registers
- A 5-bit device addressing scheme which allows the MAC to select one of up to 32 externally-connected PHY devices
- A 5-bit register addressing scheme for selecting the target register within the addressed device
- A transfer frame protocol for 16-bit read and write accesses to PHY registers via the MDC and MDIO signals under control of the MAC (PHY devices may not directly initiate MDIO transfers.)

Standard PHY control and status registers provide device capability status bits (for example, auto-negotiation, duplex modes, 10/100 speeds and protocols), device status bits (for example, auto-negotiation complete, link status, remote fault), and device control bits (for example, reset, speed selection, loopback, and auto-negotiation start).

The transfer frame protocol defines a MDC clock at a nominal period of 400ns, and an MDIO frame up to 64 bits in length. The MDIO frame consists of an optional 32-bit preamble driven by the MAC, 14 control bits driven by the MAC including the opcode and addresses, a 2-bit turn-around sequence, and a 16-bit data transfer driven either by the MAC or the PHY. Note that various PHYs support optional features such as reduced preamble or increased clock rate.

The features supported by the PHY may be determined at powerup by a MDIO read access (at default rates) of device capabilities in PHY status registers.

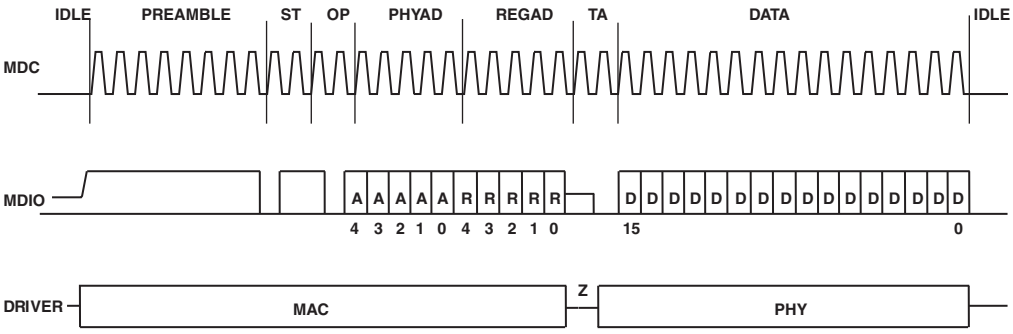


Figure 21-3. Station Management Read

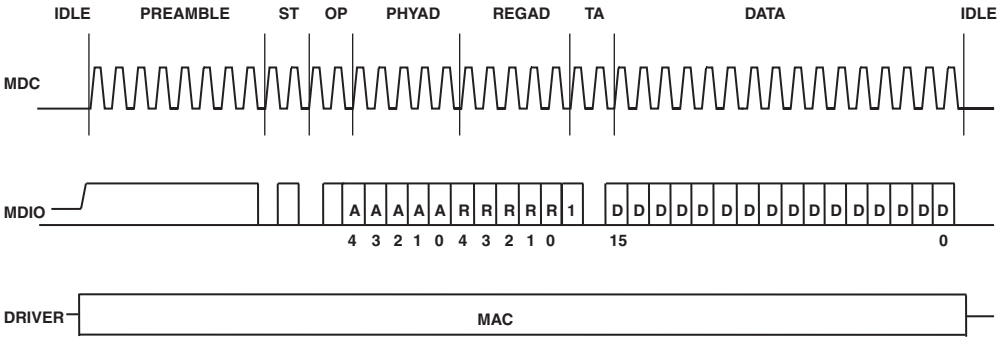


Figure 21-4. Station Management Write

Operation

The following sections describe the detailed operation of the Ethernet MAC peripheral.

MII Management Interface Operation

The MAC peripheral performs MDIO-protocol transfers in response to register read/write commands issued by the Blackfin processor. Three registers are provided to support MII management transfers:

- The EMAC_SYSCTL register contains the MDCDIV field which specifies the frequency of the MDC clock output in a ratio to the SCLK frequency, and must be initialized before any transfers.
- The EMAC_STADAT register holds the 16-bit data for read or write transfers.
- The EMAC_STAADD register supports several functions.
    - It commands the access—writes to it may initiate station management transfers, provided the STABUSY bit is set and provided that the interface is not already busy.
    - It selects the addressed device, register, and direction of the access.
    - It provides mode controls for MDIO preamble generation and station management transfer done interrupt.
    - It provides the STABUSY status bit indicating whether the interface is still busy performing a prior transfer.

As these serial accesses may require significant time (25.6us, or several thousand processor clock cycles at default rates), the Blackfin MAC provides an end-of-transfer interrupt to allow the processor to perform other functions while station management transfers are in progress. Alterna-

tively, the processor may determine the status of the transfer in progress by reading the STABUSY bit in the EMAC_STAADD register.

Receive DMA Operation

Data flow between the MAC and the Blackfin peripheral subsystem takes place via bidirectional descriptor based DMA. The element size for any DMA transfer to and from the Ethernet MAC is restricted to 32 bits. In the receive case, a queue or ring of DMA descriptor pairs are normally used, as illustrated in Figure 21-5. In the figure, data descriptors are labeled with an "A" and status descriptors are labeled with a "B."

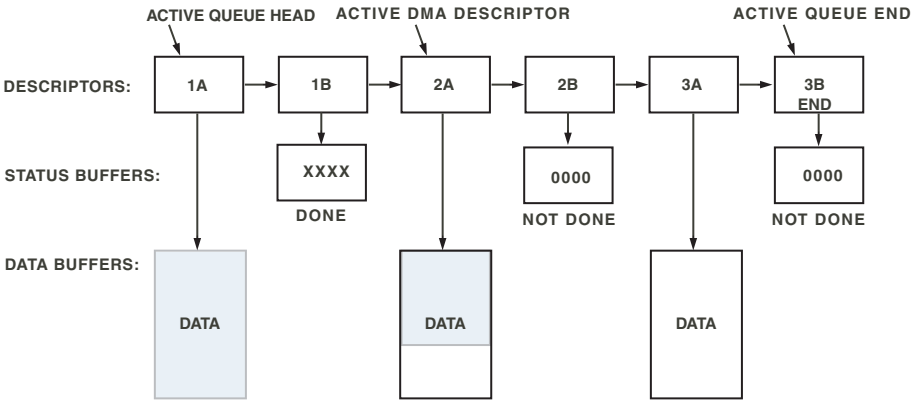


Figure 21-5. Ethernet MAC Receive DMA Operation

Receive DMA works with a queue or ring of DMA descriptor pairs structured as data and status.

- **Data** – The first descriptor in each pair points to a data buffer that is at least 1556 (0x614) bytes long and is 32-bit aligned. The descriptor XCOUNT field should be set to 0, because the MAC controls the actual buffer length.
- **Status** – The second descriptor points to a status buffer of either 4 or 8 bytes. The descriptor XCOUNT field should be set to 0, because the MAC controls the actual buffer length. After receiving and accepting any RX frame, the MAC writes a status word and optionally two IP checksum words to this status buffer. The RXCKS bit in the EMAC_SYSCTL register controls the generation of the two checksum words.

  Status words written by the MAC after frame reception have the same format as the EMAC_RX_STAT register, and always have the receive complete bit set to 1. If the driver software initializes the length/status words to 0, it can reliably interrogate (poll) an RX frame's length/status word to determine if the DMA transfer of the data buffer is complete. Alternatively, status descriptors may be individually enabled to signal an interrupt when frame reception is complete.

The MAC and DMA operate on the active queue in this manner:

- **Start** – The queue is activated by initializing the DMA next descriptor pointer and then writing the DMA_CONFIG register. Meanwhile, the MAC listens to the MII, looking for a frame that passes its address filter.
- **Data** – When a matching frame is seen, the MAC transfers the frame data into the data buffer. The MAC does not initiate the DMA transfer until either the destination address filtering is complete, or the frame ends (if a runt frame).

- **End of frame** – At the end of the frame, the MAC issues a finish command to the DMA controller, causing it to advance to the next (status) descriptor.
- **Status** – The MAC then transfers the frame status into the status buffer. The frame status structure contains the length of the frame data. The MAC then issues another finish command to complete the status DMA buffer.
- **Interrupt** – Upon completion, the DMA may issue an interrupt, if the descriptor was programmed to do so. The DMA then advances to the next (data) descriptor, if any.

Frame Reception and Filtering

Frame data written to memory normally includes the Ethernet header (destination MAC address, source MAC address, and length/type field), the Ethernet payload, and the Frame Check Sequence (FCS) checksum, but not the preamble. If the RXDWA bit in EMAC_SYSCTL is 1, then the first 16-bit word is all-zero to pad the frame. The data written includes all complete bytes for which the received data valid (ERxDV) pin on the MII interface was asserted after but not including the start of frame delimiter (SFD) nibble (1011).The preamble and any other nibbles prior to the SFD are also not included.

The MAC applies two filtering mechanisms to received frames: the address filter and the frame filter. The address filter considers only the destination MAC address and provides control over the reception of unicast, multicast, and broadcast addresses. The frame filter considers the entire frame and provides control over reception of frames with errors and of MAC control frames.

The address filter is evaluated in the following sequence. Note that this sequence is in the same order as the related bits in the EMAC_OPMODE regis-

ter, from LSB to MSB: HU, HM, PAM, PR, IFE, and DBF. The first few filter decisions are additive, while the last two are subtractive.

1. Initially, the address filter is true if the frame's MAC destination address (DA) is either the broadcast address (all 1s) or exactly matches the 48-bit station MAC address in the EMAC_ADDRHI and EMAC_ADDRLO registers.
2. **HU (hash unicast)** – If the HU bit is 1 and the DA is a unicast address which matches the hash table, the address filter is set to true.
3. **HM (hash multicast)** – If the HM bit is 1 and if the DA is a multicast address which matches the hash table, the address filter is set to true.
4. **PAM (pass all multicast)** – If the PAM bit is 1 and the DA is any multicast address, the address filter is set to true.
5. **PR (promiscuous)** – If the PR bit is 1, the address filter is set to true regardless of the frame DA.
6. **FLCE (flow control enable)** – If the FLCE bit in the EMAC_FLC register is 1, and if the DA is an exact match to either the global multicast pause address or to the station MAC address, the address filter is set to true.
7. **IFE (inverse filter)** – If the IFE bit is 1 and the DA exactly matches the 48-bit station MAC address, the address filter is set to false.
8. **DBF (disable broadcast frames)** – If the DBF bit is 1 and the DA is the broadcast address, the address filter is set to false.

The hash table address filtering is configured with the EMAC_HASHLO and EMAC_HASHHI registers described on page 21-71.

The frame filter is evaluated in the following sequence. Note that the frame filter is updated as each byte of data is received. The frame filter can

change from true to false during a frame, for example, upon DMA overrun, but can never change from false back to true.

1. Initially, the frame filter is set to true if the address filter is true, otherwise the frame filter is set to false.

2. **PCF (pass control frames)** – If the PCF bit is 0 and the frame is any valid supported MAC control frame (destination address is either the MAC address or the global multicast pause address; and the length/type field = 88-08, opcode = 0001, length = 64 bytes, and receiveOK = 1), then the frame filter is set to false.

3. **PBF (pass bad frames)** – If the PBF bit is 0 and the frame has any type of error except a frame fragment error, the frame filter is set to false. This rejects any frame for which any of these status bits are set: frame too long, alignment error, frame-CRC error, length error, or unsupported control frame. The frame filter does not reject frames on the basis of the out of range length field status bit. Note that this step may reject MAC control frames passed by PCF.

4. **PSF (pass short frames)** – If the PSF bit is 0 and the frame has a frame fragment error (frame contains less than 64 bytes), the frame filter is set to false. This step may reject frames which were passed by PCF or PBF.

5. **DMA RX overrun** – If the RX DMA FIFO overflows, the frame filter is set to false. If the FIFO overflows at a point where it contains parts of two frames, that is, the last data and status of frame A and the beginning data of frame B, then frame B is rejected by the frame filter and the MAC continues to try to deliver frame A's data and status.

Discarded Frames

Frames that fail the address filter are discarded immediately after the destination address is received, and neither their data nor their status values are written to memory via DMA. Frames that pass the address filter but fail

the frame filter before 32 bytes are received are also discarded immediately. Once at least 32 bytes of a frame have been received, and if the address and frame filters both pass, the MAC begins to write the frame to memory via DMA RX.

Aborted Frames

Frames that fail the frame filter after 32 bytes have been received are aborted. The MAC issues a restart DMA control command, causing the current RX data DMA descriptor to be reinitialized with its starting address and counts. The aborted frame's status is not written to memory. Instead, the current DMA data and status buffers are recycled for the next RX frame. For all frames that pass both the address and frame filters, both data and status are written to memory via DMA.

Control Frames

If the FLCE (flow control enable) bit is set, MAC control frames (with the control type 88-08) whose DAs match either the station MAC address (with inverse filtering disabled) or the global pause multicast address will pass the address filter, and thus may also have status of receiveOK. If the frame also is a supported pause control frame (with length = 64 bytes, and opcode = pause = 00-01, and in full-duplex mode), then the frame filter condition is determined by the PCF (pass control frames) bit. If the frame is not also a supported pause control frame, then it is in error, and its frame filter condition depends on the PBF (pass bad frames) bit.

Examples

- To perform standard IEEE-802.3 filtering, clear the EMAC_OPMODE register bits HU, PR, IFE, DBF, PBF, and PSF. With these selections, the Ethernet MAC accepts error-free broadcast frames and only those error-free unicast frames that exactly match the station MAC

address. Set PAM to accept all multicast addresses, or set HM and program the EMAC_HASHHI and EMAC_HASHLO registers to accept only a subset of multicast addresses.

- To accept all addresses, set PR and clear IFE and DBF in the EMAC_OPMODE register.
- To accept a set of several unicast addresses, set the HU bit and set the multicast hash table register bits which correspond to the desired addresses. Note that there is one set of hash table registers that apply to both unicast and multicast addresses, as selected by the HU or HM bits.
- To reject all addresses, set IFE and DBF, and clear HU, HM, PAM, and PR in the EMAC_OPMODE register.

RX Automatic Pad Stripping

If the ASTP bit in the EMAC_OPMODE register is set, the pad bytes and FCS are stripped from any IEEE-type frame which was lengthened (padded) to reach the minimum Ethernet frame length of 64 bytes. This applies to frames where the Ethernet length/type field is less than 46 bytes, since the Ethernet header and FCS add 18 bytes. When pad stripping occurs, only the first Length/Type + 14 bytes are written to memory via DMA, and the frame length reported in the EMAC_RX_STAT register and in the RX status DMA buffer will be Length/Type + 14 rather than the actual number of received bytes.

Pad bytes are never stripped from typed Ethernet frames. Typed Ethernet frames are frames with a length/type field that takes the type interpretation because it is greater than or equal to 0x600 (1536).

RX DMA Data Alignment

If the RXDWA bit in the EMAC_SYSCTL register is clear, the MAC delivers the frame data via DMA to a 32-bit-aligned buffer in memory, including the Ethernet header and FCS. Because the Ethernet header is an odd number

of 16-bit words long, this results in the frame payload being odd-aligned, which may be inconvenient for later processing.

If the RXDWA bit is set, however, the MAC prefixes one 16-bit pad word to the frame data with value 0x0000, resulting in a frame payload aligned on an even 16-bit boundary. See Figure 21-6.

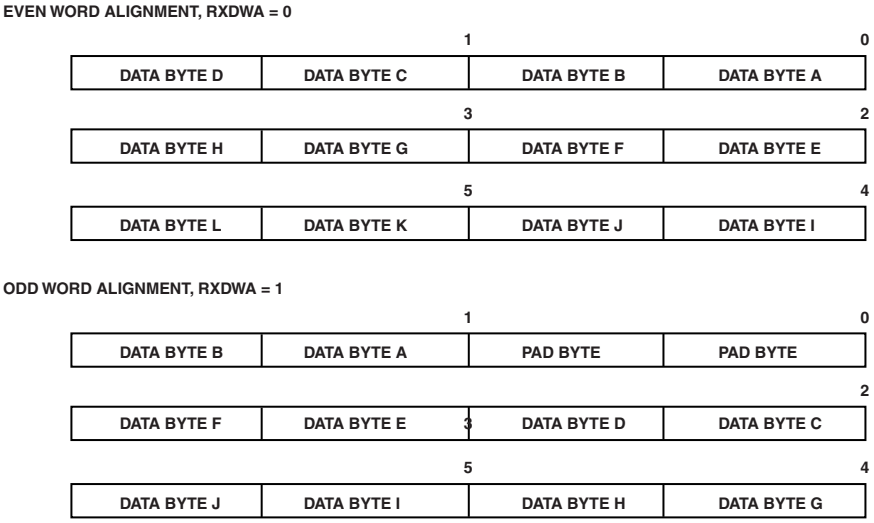


Figure 21-6. RX DMA Data Alignment

RX DMA Buffer Structure

The length of each RX DMA buffer must be at least 1556 (0x614) bytes. This is the maximum number of bytes that the MAC can deliver by DMA on any receive frame. Frames longer than the 1556-byte hardware limit are truncated by the MAC. The 1556-byte hardware limit accommodates the longest legal Ethernet frames (1518 bytes for untagged frames, or 1522 bytes for tagged 802.1Q frames) plus a small margin to accommodate future standards extensions.

The MAC does not support RX DMA data buffers composed of more than one descriptor.

RX Frame Status Buffer

The RX frame status buffer is always an integer multiple of 32-bit words in length (either 1 or 2) and must always be aligned on a 32-bit boundary. The RX frame status buffer always contains a frame status word, and may also contain two 16-bit IP checksum words if the RXCKS bit in the EMAC_SYSCTL register is set.

To synchronize RX DMA and software, the RX_COMP semaphore bit may be used in the RX frame status word. This word is always the last word written via DMA in both status buffer formats, so a transition from 0 to 1 as seen by the processor always means that both the RX data and the status buffers are entirely valid.

Table 21-2 and Table 21-3 describe each of the status buffer formats.

Table 21-2. Receive Status DMA Buffer Format (Without IP Checksum)

Offset	Size	Description
0	32	RX frame status (Same format as the EMAC_RX_STAT register)

Table 21-3. Receive Status DMA Buffer Format (With IP Checksum)

Offset	Size	Description
0	16	IP header checksum
2	16	IP payload checksum
4	32	RX frame status (Same format as the EMAC_RX_STAT register)

RX Frame Status Classification

The RX frame status buffer and the EMAC_RX_STAT register provide a convenient classification of each received frame, representing the IEEE-802.3

"receive status" code. The bit layout in the RX frame status buffer is identical to that in the EMAC_RX_STAT register, and is arranged so that exactly one status bit is asserted for each of the possible receive status codes defined in IEEE-802.3 section 4.3.2. Note in the case of a frame that does not pass the frame filter, neither the frame data nor the status are delivered by DMA into the RX frame status buffer.

The priority order for determination of the receive status code is shown in Table 21-4.

Table 21-4. RX Receive Status Priority

Priority	Bit	Bit Name	IEEE receive status	Condition
1	20	DMA overrun	Undefined	The frame was not completely delivered by DMA
2	18	Frame fragment	Not received	The frame was less than the minimum 64 bytes and was discarded without reporting any other error
3	19	Address filter failed	Not received	The frame did not pass the address filter
4	14	Frame too long	Frame too long	The frame size was more than the maximum allowable frame size (1518, 1522, or 1538 bytes for normal, VLAN1, or VLAN2 frames)
5	15	Alignment error	Alignment error	The frame did not contain an integer number of bytes, and also failed the CRC check
6	16	Frame CRC error	Frame check error	The frame failed CRC validation, and/or RX_ER was asserted during reception of the frame
7	17	Length error	Length error	The frame's length/type field was < 0x600 but did not match the actual length of the data received
8	13	Receive OK	receiveOK	The frame had none of the above conditions

RX IP Frame Checksum Calculation

The MAC calculates TCP/IP-style “raw” checksums of two useful segments of the frame data. Checksum calculation is enabled when the RXCKS bit is set to 1 in the EMAC_SYSCTL register.

The two checksum segments correspond to the typical position of the IP header and of the IP payload (see Table 21-5). The checksums are computed as a 16-bit one’s-complement sum of the selected big-endian data words. In each summand, the most significant byte is stored in byte[1] and the least significant byte is stored in byte[2], counting bytes starting at 1. If an odd number of data bytes is to be summed, the final value is stored in the most significant byte and zero is stored in the least significant byte. One’s complement addition can be done in ordinary unsigned integer arithmetic by adding the two numbers, followed by adding the carry-out bit value in at the least significant bit. This gives one’s-complement addition the property of being endian invariant, which makes it possible for software running on Blackfin’s little-endian architecture to adjust the sums without explicit byte swapping. See also *RFC 1624* and its references.

The checksum calculation hardware provides an enormous boost to TCP/IP throughput and bandwidth, but requires checksum corrections in software to properly adapt to the details of each packet protocol. For example, TCP packets require the payload checksum to include a TCP pseudo-header made up of certain fields of the IP header. These fields should be added to the “raw” hardware-generated checksum. Similarly,

the Ethernet FCS at the end of the frame should be deducted. These adjustments must be made before the IP checksum can be validated.

Table 21-5. IP Checksum Byte Ranges

Byte Number	Description	Included in IP Header Checksum?	Included in IP Payload Checksum?
1–14	Standard Ethernet header: dest address, src address, length/type	No	No
15–34	Typical IP header, without IP header options	Yes	No
35–N	IP payload, including Ethernet FCS	No	Yes

RX DMA Direction Errors

The RX DMA channel halts immediately after any transfer that sets the RXDMAERR bit in the EMAC_SYSTAT register. This bit is set if an RX data or RX status DMA request is granted by the RX DMA channel, but the DMA channel is programmed to transfer in the wrong (memory-read) direction. This could indicate a software problem in managing the RX DMA descriptor queue.

In order to facilitate software debugging, the RX DMA channel guarantees that the last transfer to occur is the one with the direction error. On an error, usually the current frame is corrupted. All later frames are ignored until the error is cleared. Since the MAC may have lost synchronization with the DMA descriptor queue, the RX channel must be disabled in order to clear the error condition.

To clear the error and resume operation, perform these steps:

1. Disable the MAC RX channel
   (clear the RE bit in the EMAC_OPCODE register).
2. Disable the DMA channel.

3. Clear the RXDMAERR bit in the EMAC_SYSSTAT register by writing 1 to it.
4. Reconfigure the MAC and the DMA engine as if starting from scratch.
5. Re-enable the DMA channel.
6. Re-enable the MAC RX channel.

Transmit DMA Operation

Figure 21-7 shows the transmit DMA operation.

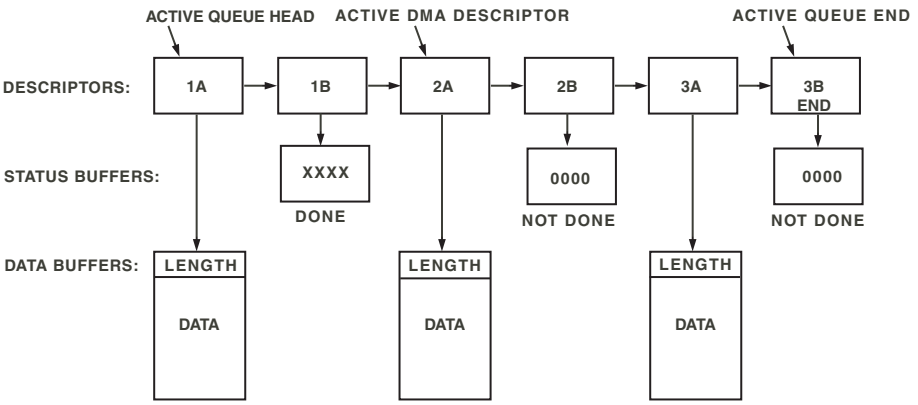


Figure 21-7. Ethernet MAC Transmit DMA Operation

Transmit DMA normally works with a queue or ring of DMA descriptor pairs.

- **Data** – The first descriptor in each pair points to a memory-read data buffer aligned on a 32-bit boundary. The first 16-bit word contains the length in bytes of the frame data, not including the length word or FCS. The descriptor `XCOUNT` field should be set to 0.
- **Status** – The second descriptor points to a 4-byte status buffer which is written via DMA at the end of the frame. The descriptor `XCOUNT` field should be set to 0, because the MAC controls the termination of the status buffer DMA. The driver software should initialize the status words to zero in advance.

  Status words written by the MAC after frame reception have the same format as the `EMAC_TX_STAT` register and always have the transmit complete bit set to 1. Software can therefore interrogate (poll) a TX frame's status word to determine if the transmission of its frame data is complete. Alternatively, status descriptors can be individually enabled to signal an interrupt when frame transmission is complete.

The MAC and DMA operate on the active queue in this manner:

- **Start** – The queue is activated by initializing the DMA `NEXT_DESC_PTR` register and then writing the `DMA_CONFIG` register.
- **Data** – The MAC transfers the frame length word and the first bytes of frame data into its TX data FIFO via DMA. When 32 bytes of data are present in the FIFO, and if the medium is unoccupied, the MAC begins transmission on the MII.
- **Collisions** – The MAC transfers data from memory via DMA into its FIFO, and then from the FIFO over the MII to the PHY. Collisions (in half-duplex mode) can occur at any time in the first 64 bytes of MII transmission, however, the MAC does not discard any of the data in its 96-byte TX FIFO until the first 64 bytes have

been successfully transmitted. If a collision occurs during this collision window, and if retry is enabled (DRTY = 0), the MAC rewinds its FIFO pointer back to the start of the frame data and begins transmission again. No redundant DMA transfers are performed in such collisions. The MAC makes up to 16 attempts to transmit the frame in response to collisions (if not disabled by DRTY), each time backing off and waiting. After the 16th attempt, the frame is aborted—the MAC terminates data transmission by sending a finish command to the DMA controller, then sending frame status, and then proceeding to the next frame data.

- **Late collisions** – After the collision window is passed, the MAC allows DMA into the FIFO to resume and to overwrite older data. If a collision occurs after the 96th byte has been transferred into the FIFO by DMA (that is, after the FIFO has "wrapped around"), then the MAC issues a restart command to the DMA controller to repeat the DMA of the current descriptor's data buffer (if enabled by the LCTRE bit).
- **End of frame** – At the end of the frame, the MAC issues a finish command to the DMA controller, causing it to advance to the next (status) descriptor. If the TX frame exceeds the maximum length limit (1560 bytes, or 0x618), the frame's DMA transfer is truncated. Only 1543 (0x607) are transmitted on the MII.
- **Status** – The MAC transfers the frame status into the status buffer.
- **Interrupt** – Upon completion, the DMA may issue an interrupt, if the descriptor was programmed to do so. The DMA then advances to the next (data) descriptor, if any.

Figure 21-8 shows an alternative descriptor structure. The frame length value and Ethernet MAC header are separated from the data payload in each frame.

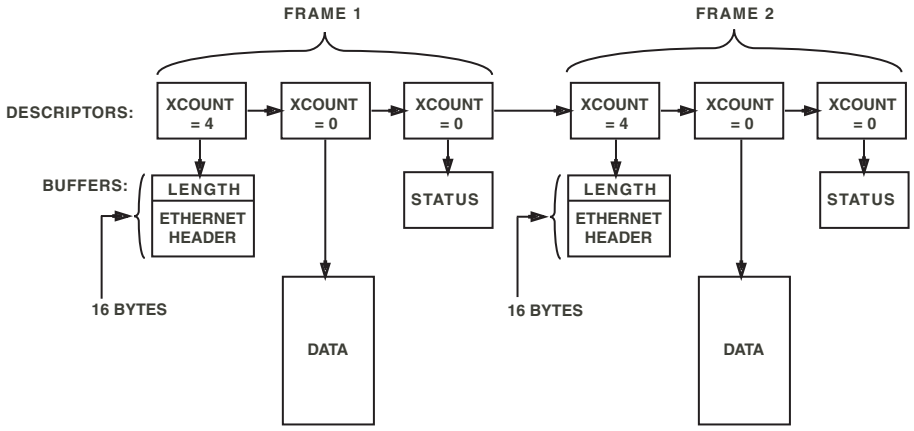


Figure 21-8. Alternative Descriptor Structure

Flexible Descriptor Structure

The Blackfin processor's DMA structure allows flexibility in the arrangement of TX frame data in memory. The frame data can be partitioned into segments, each with a separate DMA descriptor, which allows any of the first 88 bytes of DMA data (86 bytes of frame data) to reside in a separate data segment from the remainder of the frame. This permits the frame length word, the Ethernet MAC header, and even some higher level stack headers to be in one area of memory, while the payload data might be in another. The header and payload may even be in different memory spaces (some internal, some external). Each data buffer segment must be 32-bit aligned. In each frame, the `XCOUNT` field of all but the last data descriptor should be set to the actual length of the data buffers that they reference. As usual, the `XCOUNT` field of the last data descriptor should be set to 0 and the `XCOUNT` field of the status descriptor should be set to 0. The data after the first 88 bytes must all be contained in the data buffer of the last descriptor in the packet.

Multi-descriptor data formatting is not supported if retry is enabled upon late collisions (LCRTE = 1 in the EMAC_OPMODE register). The LCRTE bit must be 0 in order to use multiple DMA descriptors for transmit.

TX DMA Data Alignment

The MAC receives TX frame data via DMA from a 32-bit-aligned buffer in memory. If the TXDWA bit in the EMAC_SYSCTL register is clear, the first word of the MAC frame destination address should immediately follow the TX DMA length word. The MAC frame header starts at an odd word address and the MAC frame payload starts at an even word address.

If the TXDWA bit is set, the 16-bit TX DMA length word should be followed by a 16-bit pad word that the MAC ignores. The pad word is transferred over DMA but is not transmitted by the MAC to the PHY. The first word of the MAC frame destination address should immediately follow the pad word. The MAC frame header starts at an even word address and the MAC frame payload starts at an odd word address.

In all cases, the TX DMA length word specifies the number of bytes to be transfered via DMA, excluding the TX DMA length word itself. Specifically, when TXDWA is set, the TX DMA length word includes the length of the two pad bytes. See Figure 21-9.

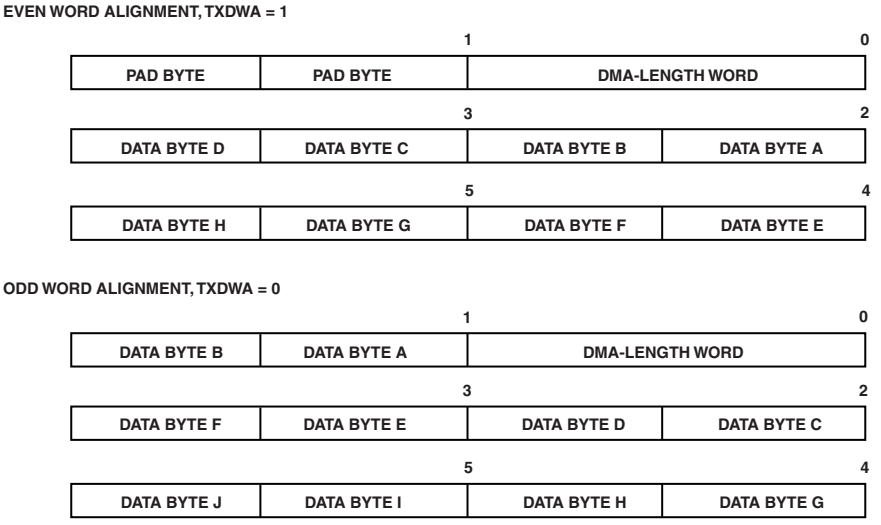


Figure 21-9. TX DMA Data Alignment

Late Collisions

If a frame's transmission is interrupted (for example, by a late collision) after the transmission of the first 64 bytes, the MAC can be programmed to either automatically retry the frame or to discard the frame. If the LCRTE bit in the EMAC_OPMODE register is set, the MAC issues a restart command to the TX DMA channel and resets the DMA current address pointer to the start of the current DMA descriptor. This requires the frame data to be entirely contained in a single DMA descriptor.

If the LCRTE bit is clear and a late collision is detected, the MAC issues a finish command to the TX DMA controller, advancing the DMA channel to the status descriptor. The MAC then transfers the TX frame status to memory and advances to the next frame descriptor for data.

TX Frame Status Classification

The TX frame status buffer and the EMAC_TX_STAT register provide a convenient classification of each received frame, representing the IEEE-802.3

"transmit status" code. The bit layout in the TX frame status buffer is identical to that in the EMAC_TX_STAT register, and is arranged so that exactly one status bit is asserted for each of the possible transmit status codes defined in IEEE-802.3 section 4.3.2.

The priority order for determination of the transmit status code is shown in Table 21-6.

Table 21-6. TX Transmit Status Priority

Priority	Bit	Bit Name	IEEE transmit status	Condition
1	4	DMA underrun	Undefined	The frame was not completely delivered by DMA
2	2	Excessive collision	Excessive collision error	The frame was aborted because of too many (16) collisions, or because of excessive deferral
3	3	Late collision error	Late collision error status	The frame was aborted because of a late collision
4	14, 13	Loss of carrier, no carrier		Carrier sense was deasserted during some or all of the frame transmission (half-duplex only, MII mode only).
5	1	Transmit OK	Transmit OK	The frame had none of the above conditions

TX DMA Direction Errors

The TX DMA channel halts immediately after any transfer that sets the TXDMAERR bit in the EMAC_SYSTAT register. This bit is set if a TX data or status DMA request is granted by the DMA channel, but the DMA channel is programmed to transfer in the wrong direction. Data DMA should be memory-read; status DMA should be memory-write. TX DMA errors could indicate a software problem in managing the TX DMA descriptor queue.

In order to facilitate software debugging, the TX DMA channel guarantees that the last transfer to occur is the one with the direction error. On an error, usually the current frame is corrupted. Any later frames in the descriptor queue are not sent until the error is cleared. Since the MAC may have lost synchronization with the DMA descriptor queue, the TX channel must be disabled in order to clear the error condition.

To clear the error and resume operation, perform these steps:

1. Disable the MAC TX channel (clear the TE bit in the EMAC_OPCODE register).
2. Disable the DMA channel.
3. Clear the TXDMAERR bit in the EMAC_SYSSTAT register by writing 1 to it.
4. Reconfigure the MAC and the DMA engine as if starting from scratch.
5. Re-enable the DMA channel.
6. Re-enable the MAC TX channel.

Power Management

The Blackfin MAC can be programmed to trigger the following two types of power state transitions:

1. Wake from hibernate

   When the processor is in hibernate state (V_{DDINT} powered off) or any higher state, a low level on the PHYINT pin can wake the processor to the full on state (via RESET). This transition is enabled by setting the PHYWE bit to 1 in the VR_CTL register prior to powerdown (See "Dynamic Supply Voltage Control" on page 18-16

This pin may be connected to an INT output of the external PHY, if applicable. Many PHY devices provide such a pin (sometimes called MDINT or INTR). PHYs with interrupt capability may be programmed in advance via the MII management interface (MDC/MDIO) to assert the INT pin asynchronously upon detecting various conditions. Examples of INT conditions include link up, remote fault, link status change, auto-negotiation complete, and duplex and speed status change.

Note that the PHYINT pin is general-purpose, and may be driven by any external device or left unused (pulled up to V_{DDEXT}). It is not limited to use with external PHYs.

When the processor is in either the hibernate or deep sleep state, the MAC is powered down. It is not possible to receive or transmit Ethernet frames in these states.

2. Wake from sleep

   When the processor is in the sleep state (or any higher state), the Ethernet MAC can remain powered up and can wake the processor to the active or full on states upon signalling an Ethernet event interrupt. The Ethernet event interrupts most useful for power management include:

   - Remote wakeup frame received, matching one of four programmable frame filters (see "Remote Wake-up Filters" on page 21-34).
   - Magic Packet™ detected (see "Magic Packet Detection" on page 21-34).
   - Any of the RX or TX frame status interrupts. Examples of these interrupts include: frame received (any frame), Broadcast frame received, VLAN1 frame received, and good frame received (which includes passing the address filters.).

For example, the MAC could be programmed to wake the system upon receiving a frame with a particular group destination address, by setting the multicast frame received interrupt enable bit in the EMAC_RX_IRQE register and by selecting the appropriate address hash bins in the EMAC_HASHLO/HI registers.

Ethernet Operation in the Sleep State

When the processor is in the sleep state, the Ethernet MAC supports several levels of operation.

- The MAC may be powered down, by clearing RE and TE in the EMAC_OPMODE register. In this lowest-power state, the MAC's internal clocks do not run, and the MAC neither transmits nor responds to received frames. Note that the MAC will not receive a PAUSE control frame in this state.
- The MAC receiver may be partially powered up in a "wake-detect-only" state, but without enabling either the MAC transmitter or MAC DMA. This state is selected by:

1. Setting RE and clearing TE in the EMAC_OPMODE register.
2. Setting either the MPKE (magic packet wake enable) or RWKE (remote wakeup frame enable) bits in the EMAC_WKUP_CTL register.
3. Clearing the capture wakeup frame (CAPWKFRM) bit in EMAC_WKUP_CTL.

   When in the wake-detect-only state, the MAC receiver disables its DMA interface, and does not request any DMA transfers (whether data or status). Instead, the MAC receiver processes good incoming frames through its remote wake-up and/or Magic Packet filters.

When a match is detected, the MAC signals a WAKEDET interrupt (setting the WAKEDET status bit in the EMAC_SYSSTAT register). DMA transfers do not resume until the CAPWKFRM bit is cleared.

- The MAC receiver may be fully powered up to both receive and/or transmit frames, provided that only external memory (for example, SDRAM) is used. Both the DMA data buffers and descriptor structures must be in external memory, since internal SRAM is unavailable when core clocks are stopped.

  This state is intended to be used with very restricted receive-frame filters, so that only certain specific frames are stored via DMA—perhaps only the frame(s) which caused the wakeup event itself. The transmit functionality permits the processor to enqueue a list of final frame transmissions before going to sleep.

  The MAC can only transmit frames contained in DMA buffers set up by the processor prior to entering the sleep state. Once the last transmit frame has been sent, the transmitter and DMA channel pauses. Note that if the last TX DMA descriptor was programmed to signal an interrupt, the processor wakes from sleep at the conclusion of that transmission.

  Similarly, the MAC can only receive as many frames as can be contained in the DMA buffers and descriptors allocated by the processor prior to entering the sleep state. Once the last receive frame has been filled, the DMA channel pauses, and if any further frames are received (beyond the capacity of the MAC RX FIFO), a DMA overrun occurs. Note that if the last RX DMA descriptor was programmed to signal an interrupt, the processor wakes from sleep after that frame was received.

Magic Packet Detection

The MAC can be programmed to detect a Magic Packet as a wakeup event. This is enabled by setting the MPKE bit (Magic Packet enable) bit in the EMAC_WKUP_CTL register. When the MAC receives the Magic Packet, it sets the MPKS (Magic Packet status) bit in the EMAC_WKUP_CTL register, which causes the Ethernet event interrupt to be asserted. The associated ISR should clear the interrupt by writing a 1 to the MPKS bit; writing a 0 has no effect.

A Magic Packet is any valid Ethernet frame which contains a specific 102-byte pattern derived from the MAC's 48-bit MAC address anywhere within the frame after the 12th byte (after the destination and source address fields). This byte pattern consists of 6 consecutive bytes of 0xFFs followed by sixteen consecutive repeats of the MAC address of the MAC which is targeted for wakeup. See Figure 21-10.

Good Magic Packet frames exclude frame-too-short error, frame-too-long error, FCS error, Alignment error, and PHY error conditions.

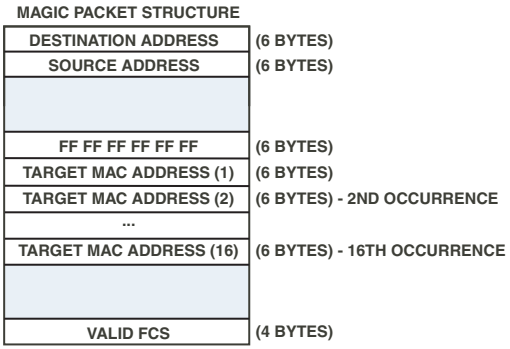


Figure 21-10. Magic Packet Structure

Remote Wake-up Filters

The Blackfin Ethernet MAC provides four independent remote wakeup frame filters for use while in powerdown. See Figure 21-11. These filters

are enabled by setting the RWKE (remote wakeup enable) bit in the EMAC_WKUP_CTL register. Each filter works in parallel, simultaneously examining each incoming frame for a specific byte pattern. Each pattern is described by a byte offset to the start of the pattern within the frame, a 32-bit byte mask selecting bytes at that offset to include in the pattern, and a CRC-16 hash value of the selected bytes which identifies the pattern.

Each of the four filters sets a separate status bit (RWKS0–RWKS3) in the EMAC_WKUP_CTL register upon detection of their programmed frame pattern. The Ethernet event interrupt is asserted when any of these four status bits is set to 1; the WAKEDET bit in the EMAC_SYSSTAT register indicates the logical OR of all four of these bits and the MPKS (Magic Packet status) bit.

The remote wakeup interrupt is cleared by writing a 1 to the appropriate RWKS0–RWKS3 status bit(s). The WAKEDET bit is read-only and does not need to be explicitly cleared.

To program each remote wakeup filter:

1. The RWKE bit in the EMAC_WKUP_CTL register must be set to 1 (enables all four filters.).
2. The enable wakeup filter N bit in the EMAC_WKUP_FFCMD register must be set to 1 to enable filter N.
3. The wakeup filter N address type bit in the EMAC_WKUP_FFCMD register selects whether the target frame is unicast (if 0) or multicast (if 1).
4. The 8-bit pattern offset N field in the EMAC_WKUP_FFOFF register selects the starting byte offset for the target data pattern, counting from 0 for the first byte of the MAC frame. The preamble and SFD bytes are not included.

5. The 32-bit EMAC_WKUP_FFMSKn register selects which of the 32 bytes starting at the selected offset into the frame will be considered in the pattern match. If the EMAC_WKUP_FFOFF register field contains the value K, then bit J of the EMAC_WKUP_FFMSKn register set controls whether byte (J+K) of the frame will be compared, counting from 0. A value of 1 in the mask bit enables comparison.
6. The 16-bit wakeup filter N pattern CRC field in the EMAC_WKUP_FFCRC0/1 register specifies the 16-bit CRC hash value expected for the wake-up pattern.

Each filter has a separate 16-bit CRC state register which is independently updated as the frame is received. The CRC state for filter N is only updated when an enabled byte is received; the CRC state remains unchanged if the current byte is not enabled by the filter's byte offset and mask registers.

Good frames whose CRC-16 value matches the specified value at the end of the selected pattern window will cause a wake-up event at the end of the frame. Good wake-up frames exclude frame-too-short error,

frame-too-long error, alignment error, FCS error, PHY error, and length error conditions.

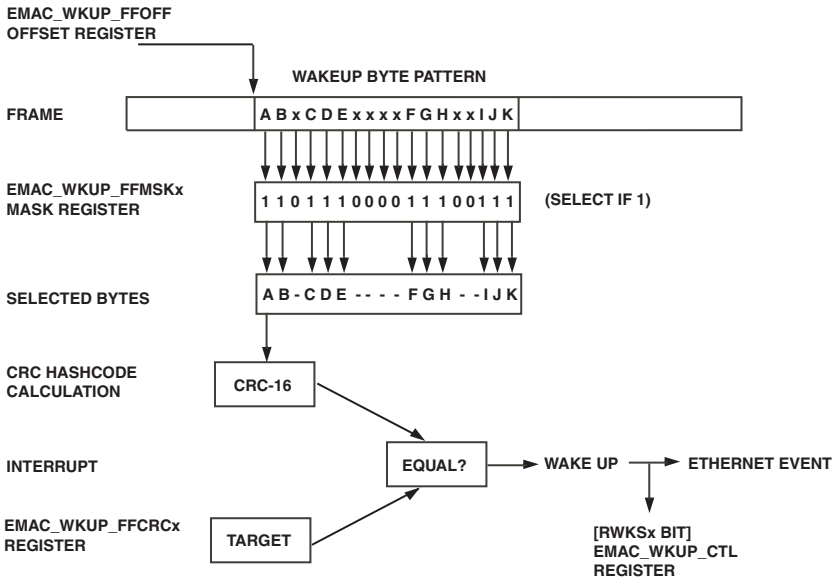


Figure 21-11. Remote Wakeup Filters

The CRC-16 hash value for a sequence of bytes may be calculated serially, with each byte processed LSB-first. The initial value of the CRC state is 0xFFFF (all 1s). For each input bit, the LFSR is shifted left one position, and the bit shifted out is XORed with the new input bit. The resulting feedback bit is then XORed into the LFSR at bit positions 15, 2, and 1. Thus the generator polynomial for this CRC is:

$$G(x) = x^{16} + x^{15} + x^2 + 1$$

For example, if the wakeup pattern specified the single byte 0x12, or 0100_1000 (LSB first), the calculation of the wakeup CRC_16 is performed as shown in Table 21-7:

G polynomial = 1000 0000 0000 0101

Table 21-7. CRC-16 Hash Value Calculation

Bit In	XOR	MSB Bit	Feedback Bit	CRC State
				1111 1111 1111 1111, Initial = 0xFFFF
0		1	1	0111 1111 1111 1011
1		0	1	0111 1111 1111 0011
0		0	0	1111 1111 1110 0110
0		1	1	0111 1111 1100 1001
1		0	1	0111 1111 1001 0111
0		0	0	1111 1111 0010 1110
0		1	1	0111 1110 0101 1001
0		0	0	1111 1100 1011 0010, Final = 0xFCB2

Ethernet Event Interrupts

The Ethernet event interrupt is signalled to indicate that any or all of the conditions listed below are pending. Figure 21-12 shows the Ethernet event interrupts.

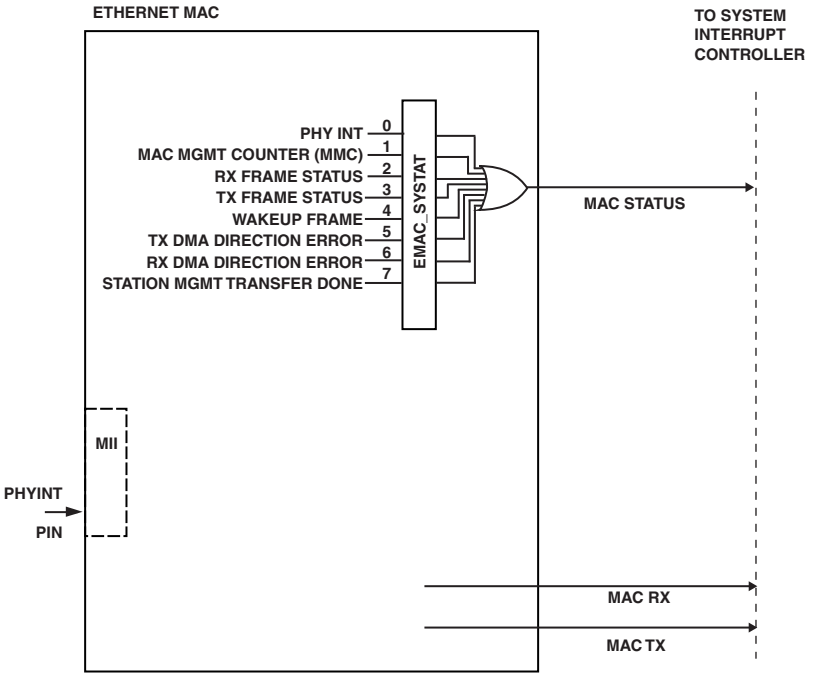


Figure 21-12. Ethernet MAC Event Interrupt

The handler for the peripheral interrupt ID corresponding to the MAC STATUS assignment should interrogate each of the peripherals assigned to that peripheral interrupt ID to determine which peripheral or peripherals are asserting an interrupt. To interrogate the Ethernet MAC, the handler should read the `EMAC_SYSTAT` register, as all of the MAC Ethernet event interrupt condition types are represented in that register.

These conditions result in an Ethernet event interrupt:

- **PHYINT interrupt** – Whenever the asynchronous PHYINT pin is asserted low, the PHYINT sticky bit in the EMAC_SYSTAT register is set to 1. The PHYINT interrupt condition is asserted whenever the logical AND of the PHYINT bit and the PHYIE enable bit in the EMAC_SYSCTL register is 1. This condition is cleared by writing a 1 to the PHYINT bit.
- **MAC management counter (MMC) interrupt** – When any MMC counter reaches half of its maximum value (that is, transitions from 0x7FFF FFFF to 0x8000 0000), the corresponding bit in the MMC RX or TX interrupt status register is set. An MMC interrupt is asserted whenever either:
  - the logical AND of the EMAC_MMC_RIRQS register and the EMAC_MMC_RIRQE register is nonzero, or
  - the logical AND of the EMAC_MMC_TIRQS register and the EMAC_MMC_TIRQE register is nonzero.

  The MMC interrupt condition is cleared by writing 1s to all of the MMC RX and/or TX interrupt status register bits which are enabled in the MMC RX/TX interrupt enable register.
- **RX frame status interrupt** – The RX frame status interrupt condition is signalled whenever the logical AND of the EMAC_RX_STKY register and the EMAC_RX_IRQE register is nonzero. This condition is cleared by writing 1s to all of the EMAC_RX_STKY register bits that are enabled in the EMAC_RX_IRQE register.
- **TX frame status interrupt** – The TX frame status interrupt condition is signalled whenever the logical AND of the EMAC_TX_STKY register and the EMAC_TX_IRQE register is nonzero. This condition is cleared by writing 1s to all of the EMAC_TX_STKY register bits that are enabled in the EMAC_TX_IRQE register.

- **Wakeup frame detected** – This bit is set when a wakeup event is detected by the MAC core (either a magic packet or a remote wakeup packet is accepted by the wakeup filters). This condition is cleared by writing a 1 to the MPKS and/or RWKS status bits in the EMAC_WKUP_CTL register.
- **RX DMA direction error detected** – This bit is set if an RX data or status DMA request is granted by the DMA channel, but the DMA is programmed to transfer in the wrong (memory-read) direction. This could indicate a software problem in managing the RX DMA descriptor queue. This interrupt is non-maskable in the MAC and must always be handled. This condition is cleared by writing a 1 to the RXDMAERR bit in the EMAC_SYSTAT register.
- **TX DMA direction error detected** – This bit is set if a TX data or status DMA request is granted by the DMA channel, but the DMA is programmed to transfer in the wrong direction. Data DMA should be memory-read, status DMA should be memory-write. This could indicate a software problem in managing the TX DMA descriptor queue. This interrupt is non-maskable in the MAC and must always be handled. This condition is cleared by writing a 1 to the TXDMAERR bit in the EMAC_SYSTAT register.
- **Station management transfer done** – This bit is set when a station management transfer (on MDC/MDIO) has completed, provided the STAIE interrupt enable control bit is set in the EMAC_STAADD register.

When the MAC DMA engine is disabled, all the MAC peripheral requests are routed directly into the interrupt controller. This can manifest itself at startup as spurious interrupts. The solution is to configure the system in such a way that the DMA controller is always enabled before the MAC peripheral.

RX/TX Frame Status Interrupt Operation

The contents of the EMAC_RX_STAT register indicate the result of the most recent frame receive operation. The register contents are updated just after the end of the frame is received on the MII and synchronized into the system clock domain.

The contents of the EMAC_RX_STKY register are updated at the same time. Each applicable bit in the EMAC_RX_STKY register is set if the corresponding bit in the EMAC_RX_STAT register is set, otherwise the bit in the EMAC_RX_STKY register keeps its prior value.

The EMAC_RX_IRQE register is continuously bitwise ANDed with the contents of the EMAC_RX_STKY register, and then all of the resulting bits are OR'ed together to produce the RX frame status interrupt condition. The state of the RX frame status interrupt condition is readable in the RXFSINT bit of the EMAC_SYSTAT register. This interrupt condition is cleared by writing 1s to all the bits in the EMAC_RX_STKY register for which corresponding bits are set in the EMAC_RX_IRQE register. Do not attempt to clear this interrupt condition by writing a 1 to the read only RXFSINT bit; such a write has no effect.

The three EMAC_TX_STAT registers (EMAC_TX_STAT, EMAC_TX_STKY, and EMAC_TX_IRQE) operate in a similar manner.

RX Frame Status Register Operation at Startup and Shutdown

After the RE bit in the EMAC_OPMODE register is cleared, the EMAC_RX_STAT register, the EMAC_RX_STKY register, and the EMAC_RX_IRQE register hold their last state. Of course, the two writable registers can still be written.

In order to not confuse status from old and new frames, the EMAC_RX_STAT register and the EMAC_RX_STKY register are automatically cleared at a 0-to-1 transition of the RE bit. The EMAC_RX_IRQE register is not cleared when the RE bit transitions from 0 to 1. It changes state only when written.

All three of these registers are cleared at system reset.

TX Frame Status Register Operation at Startup and Shutdown

After the TE bit in the EMAC_OPMODE register is cleared, the EMAC_TX_STAT register, the EMAC_TX_STKY register, and the EMAC_TX_IRQE register hold their last state. Of course, the two writable registers can still be written.

In order to not confuse status from old and new frames, the EMAC_TX_STAT register and the EMAC_TX_STKY register are automatically cleared at a 0-to-1 transition of the TE bit. The EMAC_TX_IRQE register is not cleared when the TE bit transitions from 0 to 1. It changes state only when written.

All three of these registers are cleared at system reset.

MAC Management Counters

The Blackfin Ethernet MAC provides a comprehensive set of 32-bit read-only MAC management counters, 24 for receive and 23 for transmit, in accordance with the "Layer Management for DTEs" specification in IEEE 802.3 Sec. 30.3. When enabled by setting the MMCE bit in the EMAC_MMC_CTL register, the counters are updated automatically at the conclusion of each frame. The counters may be read at any time, but may not be written. The counters can be reset to zero all at once by writing the RSTC bit to 1.

The counters can be configured to be cleared individually after each read access if the CCOR bit is set to 1. This mode guarantees that no counts are dropped between the value returned by the read and the value remaining in the register.

Although this read operation has a side effect, the speculative read operation of the Blackfin core pipeline is properly handled by the MAC. During the time between the speculative read stage and the

commit stage of the read instruction, the MMC block freezes the addressed counter so that intervening updates are deferred until the MMR read instruction is resolved.

For best results, to minimize the amount of time that any given MMC counter is frozen, it is suggested not to intentionally place MMC counter read instructions in positions that result in frequent speculative reads which are not ultimately executed. For example, MMC counter reads should not be placed in the shadow of frequently-mispredicted flow-of-control operations.

Continuous polling of any MMC register is not recommended. The MMC update process requires at least one SCLK cycle between successive reads to the same register, which may not occur if the register read is placed in a tight code loop. If the polling operation excludes the MMC update process, loss of information results.

The overflow behavior of the counters is configurable using the CROLL bit. The counters may be configured either to saturate at maximum value (CROLL = 0) or to roll over to zero and continue counting (CROLL = 1).

The range of the counters can be extended into software-managed counters (for example, 64-bit counters) by use of selectable MMC interrupts. The EMAC_MMC_RIRQE and EMAC_MMC_TIRQE registers allow the programmer to select which counters should signal an MMC interrupt on the Ethernet event interrupt line when they pass half of the maximum counter value. Even if interrupt latency is large, this mechanism makes it unlikely that any counter data is lost to overrun.

A recommended structure for the ISR for the MMC interrupt would be as follows. In this example, the CCOR (clear counter on read) bit is set to 1, and the CROLL (counter rollover) bit may also be set to 1.

1. In the ISR, read the SIC to determine which peripheral ID caused the interrupt.
2. If an Ethernet MAC event interrupt is pending, then read the EMAC_SYSTAT register. If any of the interrupt bits are set, then an Ethernet event interrupt is pending.
3. If the MMCINT bit is set, then read the EMAC_MMC_RIRQS and EMAC_MMC_TIRQS registers. Then, for each bit that is set, read the corresponding MMC counter using CCOR (clear counter on read) mode, and add the result to the software-maintained counter.

   As an option, if the CROLL bit is set to 1, the ISR can check the count value to see if it is less than 0x8000 0000. This would indicate that the counter has somehow incremented beyond the maximum value (0xFFFF FFFF) and wrapped around to zero while the interrupt awaited servicing. In this case, the software could add an additional 2^{32} to its extended counter to repair the count deficit.
4. Write the interrupt-status values previously read from EMAC_MMC_RIRQS and EMAC_MMC_TIRQS back to those same registers, so that the bits which were 1 cause the corresponding interrupt status bits to be cleared in a write-1-to-clear operation. This guarantees that all the counter interrupts that are cleared are those that correspond to counters that have been read by the interrupt handler. If other counter(s) cross the half-maximum interrupt threshold after the “snapshot” of the EMAC_MMC_RIRQS and EMAC_MMC_TIRQS was taken, then those interrupts are still correctly pending at the RTI; the interrupt handler is then re-entered and the remaining counter interrupts are handled in a second pass.

Programming Model

The following sections describe the Ethernet MAC programming model for a typical system. The initialization sequence can be summarized as follows.

1. Configure MAC MII pins.
   - Multiplexing scheme
   - CLKBUF
2. Configure interrupts.
3. Configure MAC registers.
   - MAC address
   - MII station management
4. Configure PHY.
5. Receive and transmit data through the DMA engine.

Configure MAC Pins

The first step is to configure the hardware interface between the MAC and the external PHY device.

Multiplexing Scheme

The MII interface pins are multiplexed with GPIO pins on the I/O ports. To configure a pin on these ports for Ethernet MAC functionality, see Chapter 9, “General-Purpose Ports”.

CLKBUF

The external PHY chip can be clocked with the buffered clock (CLKBUF) output from the Blackfin processor. In order to enable this clock output, the CLKBUFOE bit in the VR_CTL register must be set. Note that writes to VR_CTL take effect only after the execution of a PLL programming sequence.

Configure Interrupts

Next, the MAC interrupts and MAC DMA interrupts need to be configured properly. Interrupt service routines should be installed to handle all applicable events. Refer to Figure 21-12 on page 21-39 for a graphical representation of how event signals are propagated through the interrupt controller. The status of the MAC interrupts can be sensed with the EMAC_SYSTAT register. However, the process of enabling these interrupts is achieved through a number of different registers.

- The PHYINT interrupt is enabled by setting the PHYIE bit in the EMAC_SYSCTL register.
- The MAC management counter (MMC) interrupt can be enabled through the EMAC_MMC_RIRQE and EMAC_MMC_TIRQE registers.
- The RX frame status and TX frame status interrupts can be enabled through the EMAC_RX_IRQE and EMAC_TX_IRQE registers, respectively.
- The wakeup frame events are controlled through the EMAC_WKUP_CTL register.
- The TX DMA direction error detected and RX DMA direction error detected interrupts are non-maskable. Therefore, an interrupt service routine to handle them should always be installed.
- The station management transfer done interrupt is enabled through the STAIE bit of the EMAC_STAADD register.

The interrupts for the DMA channels corresponding to the Ethernet MAC transfers should be unmasked and a corresponding ISR should be installed if a polling technique is not used.

Configure MAC Registers

After the interrupts are set up correctly, the MAC address registers and the MII protocol must be initialized.

MAC Address

Set the MAC address by writing to the EMAC_ADDRHI and EMAC_ADDRLO registers. Since the MAC address is a unique number, it is usually stored in a non-volatile memory like a flash device. In this way, every system using the Blackfin MAC peripheral can be easily programmed with a different MAC address during mass production.

MII Station Management

The following procedure should be used to set up the MII communications protocol with the external PHY device.

To perform a station management write transfer:

1. Initialize MDCDIV in the EMAC_SYSCTL register. The frequency of the MDC clock is SCLK/[2 * (MDCDIV + 1)]. Thus MDCDIV = (SCLK_Freq / MDC_Freq)/2 - 1.

   For example, for a typical 400ns (2.5MHz) MDC rate at SCLK = 125MHz, set MDCDIV to (125MHz / 2.5MHz) / 2 -1 = 50/2-1 = 24.

2. Write the data into EMAC_STADAT.

3. Write EMAC_STAADD with the PHY address, register address, STAOP = 1, STABUSY = 1, and desired selections for preamble enable and interrupt enable.

4. Do not initiate another read or write access until STABUSY reads 0 or until the station management done interrupt (if enabled) has been received. Accesses attempted while STABUSY = 1 are discarded.

To perform a station management read transfer:

1. Initialize MDCDIV.

2. Write EMAC_STAADD with the PHY address, register address, STAOP = 0, STABUSY = 1, and desired selections for preamble enable and interrupt enable.

3. Wait either while polling STABUSY or until the station management done interrupt (if enabled) has been received. Note that subsequent accesses attempted while STABUSY = 1 are discarded. Proceed when STABUSY reads 0.

4. Read the data from EMAC_STADAT.

Configure PHY

After the MII interface is configured, the PHY can be programmed with the EMAC_STAADD and EMAC_STADAT registers. Before configuration, the PHY is usually issued a soft reset. Depending on the capabilities of the specific PHY device, the configurable options might include auto-negotiation, link speed, and whether the transfers are full-duplex or half-duplex. The PHY device may also be set up to assert an interrupt on certain conditions, such as a change of the link status.

Receive and Transmit Data

Data transferred over the MAC DMA must be handled with a descriptor-based DMA queue. Refer to Figure 21-5 on page 21-11 and Figure 21-7 on page 21-23 for a graphical representation of a receive queue and transmit queue, respectively.

An Ethernet frame header is placed in front of the payload of each data buffer. The data buffer structure is described in Table 21-8.

Table 21-8. Frame Header

Field	Size in Bytes
Frame size (Tx only)	2
Destination MAC address	6
Source MAC address	6
Length/type	2
Data Payload	Determined by the length/type field

Receiving Data

In order to receive data, memory buffers must be allocated to construct a queue of DMA data and status descriptors. If the RXDWA bit in EMAC_SYSCTL is 0, then the first item in the receive frame header is the destination MAC address. If the RXDWA bit in EMAC_SYSCTL is 1, then the first 16-bit word is all-zero to pad the frame, and the second item is the destination MAC address. The DMA engine is then configured through the DMA_CONFIG register. After the DMA is set up, the MAC receive functionality is enabled by setting the RE bit in EMAC_OPMODE. Completion can be signaled by interrupts or by polling the DMA status registers.

Transmitting Data

To transmit data, memory buffers must be allocated to construct a queue of DMA data and status descriptors. The first 16-bit word of the data buffers is written to signify the number of bytes in the frame. The DMA engine is then configured through the DMA_CONFIG register. After the DMA is set up, the MAC transmit functionality is enabled by setting the TE bit in EMAC_OPMODE. Completion can be signaled by interrupts or by polling the DMA status registers.

Ethernet MAC Register Definitions

The MAC register set is broken up into three groups corresponding to the peripheral's major system blocks:

- Control-status register group (MAC block)
- System interface register group (SIF block)
- MAC management counter register group (MMC block)

Most registers require 32-bit accesses, but certain registers have only 16 or fewer functional bits and can be accessed with either 16-bit or 32-bit MMR accesses.

Table 21-9 shows the functions of the MAC registers. MMC counter registers are found in Table 21-10 on page 21-54.

Table 21-9. MAC Register Mapping

Register Name	Function	Notes
Control-Status Register Group		
EMAC_OPMODE	MAC operating mode	Enables the Ethernet MAC transmitter.

Table 21-9. MAC Register Mapping (Continued)

Register Name	Function	Notes
EMAC_ADDRLO	MAC address low	Used with EMAC_ADDRHI to set the MAC address.
EMAC_ADDRHI	MAC address high	Used with EMAC_ADDRLO to set the MAC address.
EMAC_HASHLO	MAC multicast hash table low	Used with EMAC_HASHHI to hold the multicast hash table.
EMAC_HASHHI	MAC multicast hash table high	Used with EMAC_HASHLO to hold the multicast hash table.
EMAC_STAADD	MAC station management address	
EMAC_STADAT	MAC station management data	
EMAC_FLC	MAC flow control	
EMAC_VLAN1	MAC VLAN1 tag	
EMAC_VLAN2	MAC VLAN2 tag	
EMAC_WKUP_CTL	MAC wakeup frame control and status	
EMAC_WKUP_FFMSK 0	MAC wakeup frame 0 byte mask	
EMAC_WKUP_FFMSK 1	MAC wakeup frame 1 byte mask	
EMAC_WKUP_FFMSK 2	MAC wakeup frame 2 byte mask	
EMAC_WKUP_FFMSK 3	MAC wakeup frame 3 byte mask	
EMAC_WKUP_FFCMD	MAC wakeup frame filter commands	
EMAC_WKUP_FFOFF	MAC wakeup frame filter offsets	
EMAC_WKUP_FFCRC 0	MAC wakeup frame filter CRC0/1	
EMAC_WKUP_FFCRC 1	MAC wakeup frame filter CRC2/3	

Table 21-9. MAC Register Mapping (Continued)

Register Name	Function	Notes
System Interface Register Group		
EMAC_SYSCTL	MAC system control	
EMAC_SYSTAT	MAC system status	
EMAC_RX_STAT	Ethernet MAC RX current frame status	
EMAC_RX_STKY	Ethernet MAC RX sticky frame status	
EMAC_RX_IRQE	Ethernet MAC RX frame status interrupt enable	
EMAC_TX_STAT	Ethernet MAC TX current frame status	
EMAC_TX_STKY	Ethernet MAC TX sticky frame status	
EMAC_TX_IRQE	Ethernet MAC TX frame status interrupt enable	
EMAC_MMC_RIRQS	Ethernet MAC MMC RX interrupt status	
EMAC_MMC_RIRQE	Ethernet MAC MMC RX interrupt enable	
EMAC_MMC_TIRQS	Ethernet MAC MMC TX interrupt status	
EMAC_MMC_TIRQE	Ethernet MAC MMC TX interrupt enable	
MAC Management Counter Register Group		
EMAC_MMC_CTL	MAC management counters control	For a list of the MMC counter registers, see Table 21-10.

Table 21-10. MAC Management Counter Registers

Register Name (IEEE Name) IEEE 802.3 Reference	Description
EMAC_RXC_OK (FramesReceivedOK) 30.3.1.1.5	Holds a count of frames that are successfully received. This does not include frames received with frame-too-long, FCS, length or alignment errors, or frames lost due to internal MAC sublayer (DMA/FIFO) errors. This also excludes frames with frame-too-short errors, or frames that do not pass the address filter as indicated by the receive frame accepted status bit. Such frames are not considered to be received by the station, and are not considered errors.
EMAC_RXC_FCS (FrameCheckSequenceErrors) 30.3.1.1.6	Holds a count of receive frames that are an integral number of octets in length and do not pass the FCS check. This does not include frames received with frame-too-long or frame-too-short (frame fragment) errors. This also excludes frames with frame-too-short errors, or which do not pass the address filter.
EMAC_RXC_ALIGN (AlignmentErrors) 30.3.1.1.7	Holds a count of frames that are not an integral number of octets in length and do not pass the FCS check. This counter is incremented when the receive status is reported as alignment error. This also excludes frames with frame-too-short errors, or which do not pass the address filter.
EMAC_RXC_OCTET (OctetsReceivedOK) 30.3.1.1.14	Holds a count of data and padding octets in frames that are successfully received. This does not include octets in frames received with frame-too-long, FCS, length or alignment errors, or frames lost due to internal MAC sublayer errors. This also excludes frames with frame-too-short errors, or which do not pass the address filter.
EMAC_RXC_DMAOVF (FramesLostDueToIntMAC RcvError) 30.3.1.1.15	Holds a count of frames that would otherwise be received by the station, but could not be accepted due to an internal MAC sublayer receive error. If this counter is incremented, then none of the other receive counters are incremented. This counts frames truncated during DMA transfer to memory, as indicated by the DMA overrun status bit.
EMAC_RXC_UNICST (UnicastFramesReceivedOK) No IEEE reference	Holds a count of frames counted by the EMAC_RXC_OK register that are not counted by the EMAC_RXC_MULTI or the EMAC_RXC_BROAD register.

Table 21-10. MAC Management Counter Registers (Continued)

Register Name (IEEE Name) IEEE 802.3 Reference	Description
EMAC_RXC_MULTI (MulticastFramesReceivedOK) 30.3.1.1.21	Holds a count of frames that are successfully received and are directed to an active non-broadcast group address. This does not include frames received with frame-too-long, FCS, length or alignment errors, or frames lost due to internal MAC sublayer error. This also excludes frames with frame-too-short errors, or that do not pass the address filter.
EMAC_RXC_BROAD (BroadcastFramesReceivedOK) 30.3.1.1.22	Holds a count of frames that are successfully received and are directed to the broadcast group address. This does not include frames received with frame-too-long, FCS, length or alignment errors, or frames lost due to internal MAC sub-layer error. This also excludes frames with frame-too-short errors, or that do not pass the address filter.
EMAC_RXC_LNERRI (InRangeLengthErrors) 30.3.1.1.23	Holds a count of frames with a length/type field value between the minimum unpadded MAC client data size and the maximum allowed MAC client data size, inclusive, that does not match the number of MAC client data octets received. The counter also increments when a frame has a length/type field value less than the minimum allowed unpadded MAC client data size and the number of MAC client data octets received is greater than the minimum unpadded MAC client data size. This also excludes frames with frame-too-short errors (less than the minimum unpadded MAC client data size), or that do not pass the address filter.
EMAC_RXC_LNERRO (OutOfRangeLengthField) 30.3.1.1.24	Holds a count of frames with a Length field value greater than the maximum allowed LLC data size. This also excludes frames with frame-too-short errors, or that do not pass the address filter.
EMAC_RXC_LONG (FrameTooLongErrors) 30.3.1.1.25	Holds a count of frames received that exceed the maximum permitted frame size. This counter is incremented when the status of a frame reception is “frame too long.” This also excludes frames with frame-too-short errors, or that do not pass the address filter.

Table 21-10. MAC Management Counter Registers (Continued)

Register Name (IEEE Name) IEEE 802.3 Reference	Description
EMAC_RXC_MACCTL (MACControlFramesReceived) 30.3.3.4	Holds a count of MAC control frames passed by the MAC sublayer to the MAC control sublayer. This counter is incremented upon receiving a valid frame with a Length/Type field value equal to 88-08. While the control frame may be received by the Ethernet MAC and yet not be delivered to the MAC client by DMA, depending on the state of the PCF bit, the control frame is still counted by this counter.
EMAC_RXC_OPCODE (UnsupportedOpcodesReceived) 30.3.3.5	Holds a count of MAC control frames received that contain an opcode that is not supported by the device. This counter is incremented when a receive frame function call returns a valid frame with a length/type field value equal to the reserved type, and with an opcode for a function that is not supported by the device. Only opcode 00-01(pause) is supported by the Ethernet MAC.
EMAC_RXC_PAUSE (PAUSEMACCtrlFramesReceived) 30.3.4.3	Holds a count of MAC control frames passed by the MAC sublayer to the MAC control sublayer. This counter is incremented when a receive frame function call returns a valid frame with both a length/type field value equal to 88-08 and an opcode indicating the pause operation (00-01). This counter does not include or exclude frames on the basis of address, even though pause frames are required to contain the MAC control pause multicast address.
EMAC_RXC_ALLFRM (FramesReceivedAll) No IEEE reference	Holds a count of all frames or frame fragments detected by the Ethernet MAC, regardless of errors and regardless of address, except for DMA overrun frames.
EMAC_RXC_ALLOCT (OctetsReceivedAll) No IEEE reference	Holds a count of all octets in frames or frame fragments detected by the Ethernet MAC, regardless of errors and regardless of address, except for DMA overrun frames.
EMAC_RXC_TYPED (TypedFramesReceived) No IEEE reference	Holds a count of all frames received with a length/type field greater than or equal to 0x600. This does not include frames received with frame-too-long, frame-too-short, FCS, length or alignment errors, frames lost due to internal MAC sublayer error, or that do not pass the address filter.

Table 21-10. MAC Management Counter Registers (Continued)

Register Name (IEEE Name) IEEE 802.3 Reference	Description
EMAC_RXC_SHORT (FramesLenLt64Received) No IEEE reference	Holds a count of all frame fragments detected with frame-too-short errors (length < 64 bytes), regardless of address filtering or of any other errors in the frame.
EMAC_RXC_EQ64 (FramesLenEq64Received) No IEEE reference	Holds a count of all good frames (with status receiveOK) that have a length of exactly 64 bytes.
EMAC_RXC_LT128 (FramesLen65_127Received) No IEEE reference	Holds a count of all good frames (with status receiveOK) that have a length between 65 and 127 bytes, inclusive.
EMAC_RXC_LT256 (FramesLen128_255Received) No IEEE reference	Holds a count of all good frames (with status receiveOK) that have a length between 128 and 255 bytes, inclusive.
EMAC_RXC_LT512 (FramesLen256_511Received) No IEEE reference	Holds a count of all good frames (with status receiveOK) that have a length between 256 and 511 bytes, inclusive.
EMAC_RXC_LT1024 (FramesLen512_1023Received) No IEEE reference	Holds a count of all good frames (with status receiveOK) that have a length between 512 and 1023 bytes, inclusive.
EMAC_RXC_GE1024 (FramesLen1024_MaxReceived) No IEEE reference	Holds a count of all good frames (with status receiveOK) that have a length greater than or equal to 1024 bytes. This does not include frames with a frame-too-long error.
EMAC_TXC_OK (FramesTransmittedOK) 30.3.1.1.2	Holds a count of frames that are successfully transmitted. This counter is incremented when the transmit status is reported as transmit OK.
EMAC_TXC_1COL (SingleCollisionFrames) 30.3.1.1.3	Holds a count of frames that are involved in a single collision and are subsequently transmitted successfully. This counter is incremented when the result of a transmission is reported as transmit OK and the attempt value is 2.
EMAC_TXC_GT1COL (MultipleCollisionFrames) 30.3.1.1.4	Holds a count of frames that are involved in more than one collision and are subsequently transmitted successfully. This counter is incremented when the transmit status is reported as transmit OK and the value of the attempts variable is greater than 2 and less then or equal to 16.

Table 21-10. MAC Management Counter Registers (Continued)

Register Name (IEEE Name) IEEE 802.3 Reference	Description
EMAC_TXC_OCTET (OctetsTransmittedOK) 30.3.1.1.8	Holds a count of data and padding octets in frames that are successfully transmitted. This counter is incremented when the transmit status is reported as transmit OK.
EMAC_TXC_DEFER (FramesWithDeferredXmissions) 30.3.1.1.9	Holds a count of frames whose transmission was delayed on its first attempt because the medium was busy (that is, at the start of frame, CRS is asserted, or was previously asserted within the minimum interframe gap). Frames involved in any collisions are not counted.
EMAC_TXC_LATECL (LateCollisions) 30.3.1.1.10	Holds a count of times that a collision has been detected later than one slot time from the start of the frame transmission. A late collision is counted twice, both as a collision and as a late collision. This counter is incremented when the number of late collisions detected in transmission of any one frame is nonzero.
EMAC_TXC_XS_COL (FramesAbortedDueToXSColls) 30.3.1.1.11	Holds a count of frames that are not transmitted successfully due to excessive collisions. This counter is incremented when the number of attempts equals 16 during a transmission. Note this does not include frames that are successfully transmitted on the last possible attempt.
EMAC_TXC_DMAUND (FramesLostDueToIntMACXmit Error) 30.3.1.1.12	Holds a count of frames that would otherwise be transmitted by the station, but could not be sent due to an internal MAC sublayer transmit error. If this counter is incremented, then none of the other transmit counters are incremented. This counts frames whose transmission is interrupted by incomplete DMA transfer from memory, as indicated by the DMA underrun status bit.
EMAC_TXC_CRSERR (CarrierSenseErrors) 30.3.1.1.13	Holds a count of the number of times that carrier sense was not asserted or was deasserted during the transmission of a frame without collision.
EMAC_TXC_UNICST (UnicastFramesXmittedOK) No IEEE reference	Holds a count of frames counted by the `EMAC_TXC_OK` register that are not counted by the `EMAC_TXC_MULTI` or the `EMAC_TXC_BROAD` register.

Table 21-10. MAC Management Counter Registers (Continued)

Register Name (IEEE Name) IEEE 802.3 Reference	Description
EMAC_TXC_MULTI (MulticastFramesXmittedOK) 30.3.1.1.18	Holds a count of frames that are successfully transmitted to a group destination address other than broadcast.
EMAC_TXC_BROAD (BroadcastFramesXmittedOK) 30.3.1.1.19	Holds a count of frames that are successfully transmitted to the broadcast address as indicated by the transmit status of OK.
EMAC_TXC_XS_DFR (FramesWithExcessiveDeferral) 30.3.1.1.20	Holds a count of frames that deferred for an excessive period of time. This counter can only be incremented once per LLC transmission.
EMAC_TXC_MACCTL (MACControlFramesTransmitted) 30.3.3.3	Holds a count of MAC control frames passed to the MAC sublayer for transmission. Note this counter is incremented only when a MAC pause frame is generated by writing to the EMAC_FLC register. The counter is not incremented for frames transmitted via the normal DMA mechanism which happen to contain valid MAC pause data.
EMAC_TXC_ALLFRM (FramesTransmittedAll) No IEEE reference	Holds a count of all frames whose transmission has been attempted, regardless of success. Each frame is counted only once, regardless of the number of retry attempts.
EMAC_TXC_ALLOCT (OctetsTransmittedAll) No IEEE reference	Holds a count of all octets in all frames whose transmission has been attempted, regardless of success. Each frame's length is counted only once, regardless of the number of retry attempts.
EMAC_TXC_EQ64 (FramesLenEq64Transmitted) No IEEE reference	Holds a count of all frames with status transmit OK that have a length of exactly 64 bytes.
EMAC_TXC_LT128 (FramesLen65_127Transmitted) No IEEE reference	Holds a count of all frames transmitted with status transmit OK that have a length between 65 and 127 bytes, inclusive.
EMAC_TXC_LT256 (FramesLen128_255Transmitted) No IEEE reference	Holds a count of all frames transmitted with status transmit OK that have a length between 128 and 225 bytes, inclusive.

Table 21-10. MAC Management Counter Registers (Continued)

Register Name (IEEE Name) IEEE 802.3 Reference	Description
EMAC_TXC_LT512 (FramesLen256_511Transmitted) No IEEE reference	Holds a count of all frames transmitted with status transmit OK that have a length between 256 and 511 bytes, inclusive.
EMAC_TXC_LT1024 (FramesLen512_1023Transmitted) No IEEE reference	Holds a count of all frames transmitted with status transmit OK that have a length between 512 and 1023 bytes, inclusive.
EMAC_TXC_GE1024 (FramesLen1024_MaxTransmitted) No IEEE reference	Holds a count of all frames transmitted with status transmit OK that have a length greater than or equal to 1024 bytes but not greater than the maximum frame size.
EMAC_TXC_ABORT (TxAbortedFrames) No IEEE reference	Holds a count of all frames attempted that were not successfully transmitted with status of transmit OK.

Control-Status Register Group

This set of registers is used by the application software to configure and monitor the functionality of the MAC block.

MAC Operating Mode (EMAC_OPMODE) Register

The EMAC_OPMODE register, shown in Figure 21-13, controls the address filtering and collision response characteristics of the Ethernet controller in both the RX and TX modes.

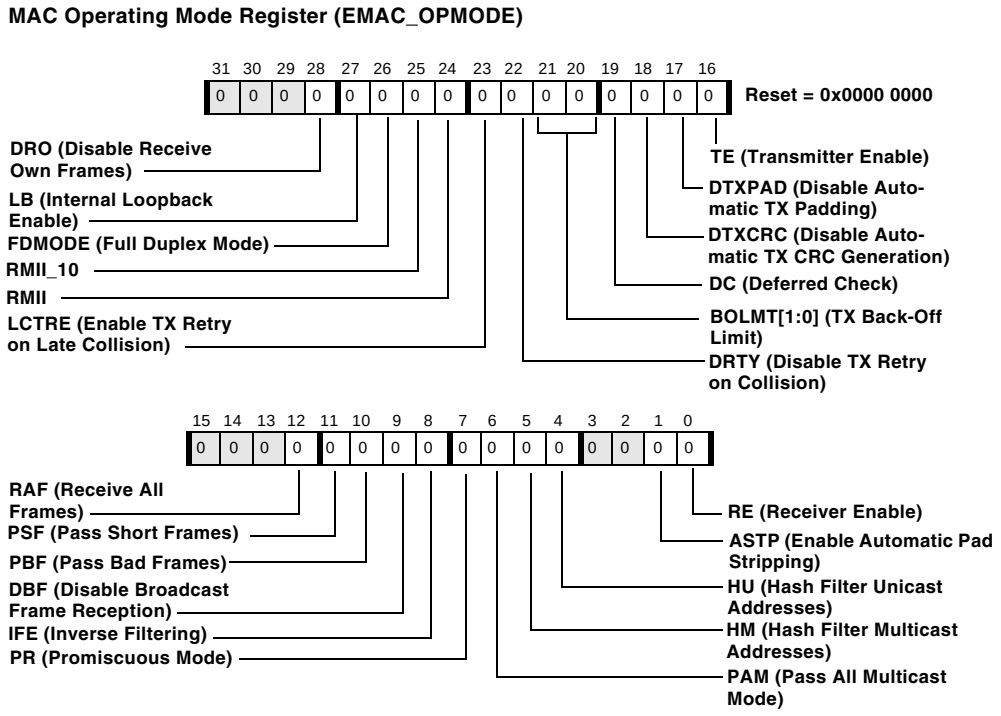

MAC Operating Mode Register (EMAC_OPMODE)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
DRO (Disable Receive Own Frames)
LB (Internal Loopback Enable)
FDMODE (Full Duplex Mode)
RMII_10
RMII
LCTRE (Enable TX Retry on Late Collision)
TE (Transmitter Enable)
DTXPAD (Disable Automatic TX Padding)
DTXCRC (Disable Automatic TX CRC Generation)
DC (Deferred Check)
BOLMT[1:0] (TX Back-Off Limit)
DRTY (Disable TX Retry on Collision)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RAF (Receive All Frames)
PSF (Pass Short Frames)
PBF (Pass Bad Frames)
DBF (Disable Broadcast Frame Reception)
IFE (Inverse Filtering)
PR (Promiscuous Mode)
RE (Receiver Enable)
ASTP (Enable Automatic Pad Stripping)
HU (Hash Filter Unicast Addresses)
HM (Hash Filter Multicast Addresses)
PAM (Pass All Multicast Mode)

Figure 21-13. EMAC_OPMODE Register

Additional information for the EMAC_OPMODE register bits includes:

- **Disable receive own frames** (DRO)

  When set in half-duplex mode, this bit blocks all frames transmitted by the MAC from being read into the receive path. This bit should be reset when the MAC is operating in full-duplex mode, MII mode only.

  0 – Receive own frames enabled

  1 – Receive own frames disabled

- **Internal loopback enable** (LB)

  When internal loopback is enabled, the frames transmitted by the MAC are internally redirected to the receive MAC port. Loopback operation is supported in MII mode; loopback is not supported in RMII mode. During loopback, the external MII port is inactive, the RX pins are ignored and the TX pins are set to TXEN = 0, TXD = 1111.

  0 – Internal loopback not enabled

  1 – Internal loopback enabled

- **Full duplex mode** (FDMODE)

  0 – Half duplex mode selected

  1 – Full duplex mode selected

- **RMII port speed selector** (RMII_10)

  When the interface is configured for RMII operation, software must query the PHY after any automatic negotiation to determine the link speed, and set the RMII port speed selector accordingly. This is because in RMII mode, the REFCLK input is always a con-

stant speed regardless of link speed. In MII mode, by contrast, the PHY decreases the speed of the RXCLK and TXCLK to 2.5 MHz when the link speed is 10M bits.

0 – Speed for RMII port is 100M bits

1 – Speed for RMII port is 10M bits

- **RMII mode** (RMII)

  This bit is used to select which interface, RMII or MII, is used by the MAC to transfer data to and from the external PHY. Note that MII and RMII modes use slightly different sets of package pins. Program different values into the port configuration register(s) accordingly.

  0 – MII mode

  1 – RMII mode

- **Enable TX retry on late collision** (LCTRE)

  0 – TX retry on late collision not enabled

  1 – TX retry on late collision enabled

- **Disable TX retry on collision** (DRTY)

  0 – TX retry on collision not disabled

  1 – TX retry on collision disabled

- **TX back-off limit** (BOLMT[1:0])

  This field sets an upper bound on the random back-off interval time before the MAC resends a packet in the event of a collision. The bound can be set to 1, 15, 255, or 1023 slot times (1 slot time = 128 MII clock cycles). Thus, varying levels of aggressiveness with regard to packet re-transmission can be selected.

00 – The number of bits is 10 and the maximum back-off time is 1023 slots (relaxed, standard-compliant behavior).

01 – The number of bits is 8 and the maximum back-off time is 255 slots.

10 – The number of bits is 4 and the maximum back-off time is 15 slots.

11 – The number of bits is 1 and the maximum back-off time is 1 slot (aggressive)

- **Deferral check** (DC)

  In half-duplex operation, a frame whose transmission defers to incoming traffic for longer than two maximum-length frame times is considered to have been excessively deferred. This time is (2 x 1518 x 2) = 6072 MII clocks. See IEEE 802.3 section 5.2.4.1 for more information.

  0 – The MAC cannot abort transmission of frames due to excessive deferral.

  1 – Enables the MAC to abort transmission of frames that encounter excessive deferral.

- **Disable automatic TX CRC generation** (DTXCRC)

  0 – Automatic TX CRC generation is enabled. Four CRC bytes are appended to the frame data.

  1 – Automatic TX CRC generation is disabled.

- **Disable automatic TX padding** (DTXPAD)

  0 – Automatic TX padding is enabled. Pad bytes with value 0 are appended to the data, followed by the CRC, so that the minimum frame size is 64 bytes.

1 – Automatic TX padding of frames shorter than 64 bytes is disabled.

- **Transmitter enable** (TE)

  The MAC transmitter is reset when TE is 0. A rising (0 to 1) transition on TE causes the EMAC_TX_STAT register and the EMAC_TX_STKY register to be reset. TE and RE may be enabled either individually or together in either MII or RMII mode.

- **Receive all frames** (RAF)

  0 – Does not override filters.

  1 – Overrides the address and frame filters and causes all frames or frame fragments to be transferred to memory by DMA.

- **Pass short frame** (PSF)

  0 – The frame filter rejects frames with frame-too-short errors (runt frames, or frames with total length less than 64 bytes not including preamble).

  1 – Short frames are not rejected by the frame filter.

- **Pass bad frames** (PBF)

  0 – The frame filter rejects frames with FCS errors, alignment errors, length errors, frame-too-long errors, and DMA overrun errors.

  1 – Pass bad frames enabled.

- **Disable broadcast frame reception** (DBF)

  0 – Broadcast frame reception not disabled.

  1 – Removes the broadcast address (all 1s) from the set of addresses passed by the address filter, overriding promiscuous mode.

- **Inverse filtering** (IFE)

  0 – Inverse filtering not enabled

  1 – Removes the MAC address programmed in the EMAC_ADDRHI and EMAC_ADDRLO registers from the set of addresses passed by the address filter, overriding PR (promiscuous) and HU (hash unicast) modes. The effect is to block reception of a specific destination address.

- **Promiscuous mode** (PR)

  0 – Promiscuous mode not enabled

  1 – Promiscuous mode enabled, the address filter accepts all addresses.

- **Pass all multicast mode** (PAM)

  0 – Do not pass all multicast frames.

  1 – All multicast frames are added to the set of addresses passed by the address filter.

- **Hash filter multicast addresses** (HM)

  0 – Does not add multicast addresses that match the hash table to the set of addresses passed by the address filter.

  1 – Adds multicast addresses that match the hash table to the set of addresses passed by the address filter.

- **Hash filter unicast addresses** (HU)

  0 – Does not add unicast addresses that match the hash table to the set of addresses passed by the address filter.

  1 – Adds unicast addresses that match the hash table to the set of addresses passed by the address filter.

- **Automatic pad stripping enable** (ASTP)

  A received frame contains pad bytes if it is in IEEE format (the length/type field contains a length value < 0x600) and if the length value is less than 46 (corresponding to a frame whose total length including header and FCS is less than 64 bytes). If ASTP = 1, both the pad and the FCS bytes are removed from the received data.

  0 – Automatic pad stripping is not enabled.

  1 – Automatic pad stripping is enabled.

- **Receiver enable** (RE)

  The MAC transmitter is reset when RE is 0. A rising (0 to 1) transition on RE causes the EMAC_RX_STAT register and the EMAC_RX_STKY register to be reset. RE and TE may be enabled either individually or together in either MII or RMII mode.

MAC Address Low (EMAC_ADDRLO) Register

The EMAC_ADDRLO register, shown in Figure 21-14, holds the low part of the unique 48-bit station address of the MAC hardware. Writes to this register must be performed while the MAC receive and transmit paths are both disabled. The byte order of address transfer is lowest significant byte first and lowest significant bit first on the MII. Thus EMAC_ADDRLO[3:0] is the first nibble transferred and EMAC_ADDRHI[15:12] is the last nibble.

For example, the address 00:12:34:56:78:9A (where 00 is transferred first and 9A is transferred last) would be programmed as:

EMAC_ADDRLO = 0x56341200

EMAC_ADDRHI = 0x00009A78

MAC Address Low Register (EMAC_ADDRLO)
R/W, except cannot be written if RX or TX is enabled in the EMAC_OPMODE register.

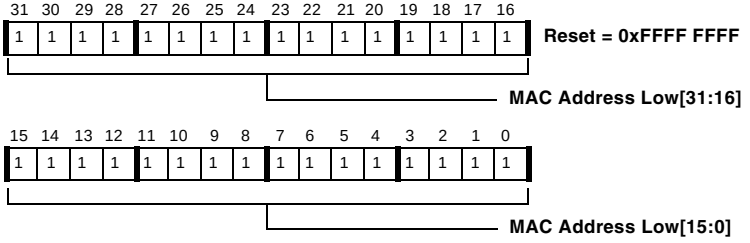


Figure 21-14. EMAC_ADDRLO Register

MAC Address High Register (EMAC_ADDRHI) Register

The EMAC_ADDRHI register, shown in Figure 21-15, holds the high part of the unique 48-bit station address of the MAC hardware. Writes to this register must be performed while the MAC receive and transmit paths are both disabled.

MAC Address High Register (EMAC_ADDRHI)
R/W, except cannot be written if RX or TX is enabled in the EMAC_OPMODE register.

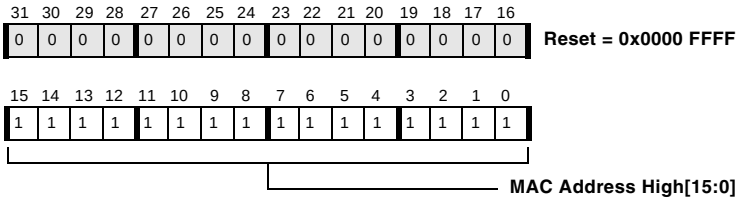


Figure 21-15. EMAC_ADDRHI Register

MAC Multicast Hash Table High (EMAC_HASHHI) and Low (EMAC_HASHLO) Registers

The EMAC_HASHLO register holds the values for bins 31–0 of the multicast hash table. The EMAC_HASHHI register holds the values for bins 63–32 of the multicast hash table. See Figure 21-16 and Figure 21-17.

The 64-bit multicast table is used for multicast frame address filtering. A cyclic redundancy check (CRC) based hash table scheme is used. After the destination address (6th byte) of the frame is received, the state of the CRC-32 checksum unit is sampled. This CRC-32 unit implements the IEEE 802.3 CRC algorithm used in validating the FCS field of the frame. The 6 most significant bits from this state identify one of 64 hash bins representing the frame's destination address. These 6 bits are then used to index into the two hash table registers and extract the corresponding hash bin enable bit. The most significant bit of this value determines the register to be used (high/low) while the other five bits determine the bit position within the register. A CRC value of 000000 selects bit 0 of the EMAC_HASHLO register and a CRC value of 111111 selects bit 31 of the EMAC_HASHHI register.

If the corresponding bit in the hash table register is set, the multicast frame is accepted. Otherwise, it is rejected. If the PM bit in the EMAC_OPMODE register is set, all multicast frames are accepted regardless of the hash values.

For example, consider the calculation of the hash bin for the MAC address 01.23.45.67.89.AB. The CRC algorithm uses an LFSR with the prime generator polynomial specified in IEEE 802.3 Sec 3.2.8:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$$

The bits of the MAC address are fed in left-most byte first, least significant bit first, in this sequence (left to right):

1000 0000 1100 0100 1010 0010 1110 0110 1001 0001 1101 0101

The 32-bit CRC register is initialized to all 1s. Then each input bit is processed as follows: first, the register is shifted left one place, shifting in a zero and shifting out the former MSB. The bit just shifted out is XORed with the current input bit, yielding the feedback bit. If this feedback bit is a 1, then the shift register contents are XORed with the generator polynomial value:

0x04C1 1DB7 = 0000 0100 1100 0001 0001 1101 1011 0111

Following this procedure, the CRC-32 for the MAC address is calculated. See Table 21-11.

Table 21-11. CRC-32 Calculation

Bit Number	Input Bit	MSB Bit	Feedback Bit	Next CRC Shift Register
Start				1111 1111 1111 1111 1111 1111 1111 1111
0	1	1	0	1111 1111 1111 1111 1111 1111 1111 1110
1	0	1	1	1111 1011 0011 1110 1110 0010 0100 1011
2	0	1	1	1111 0010 1011 1100 1101 1001 0010 0001
3	0	1	1	1110 0001 1011 1000 1010 1111 1111 0101
4	0	1	1	1100 0111 1011 0000 0100 0010 0101 1101
5	0	1	1	1000 1011 1010 0001 1001 1001 0000 1101
6	0	1	1	0001 0011 1000 0010 0010 1111 1010 1101
7	0	0	0	0010 0111 0000 0100 0101 1111 0101 1010
...				
46	0	1	1	1101 0011 1001 0111 1111 0100 0100 1001
47	1	1	0	1010 0111 0010 1111 1110 1000 1001 0010

The resulting six MSBs are 101001 = 0x29 = 41 decimal. The hash bin enable bit for this address is then bit 41 – 32 = 9 of the EMAC_HASHHI register.

MAC Multicast Hash Table Low Register (EMAC_HASHLO)

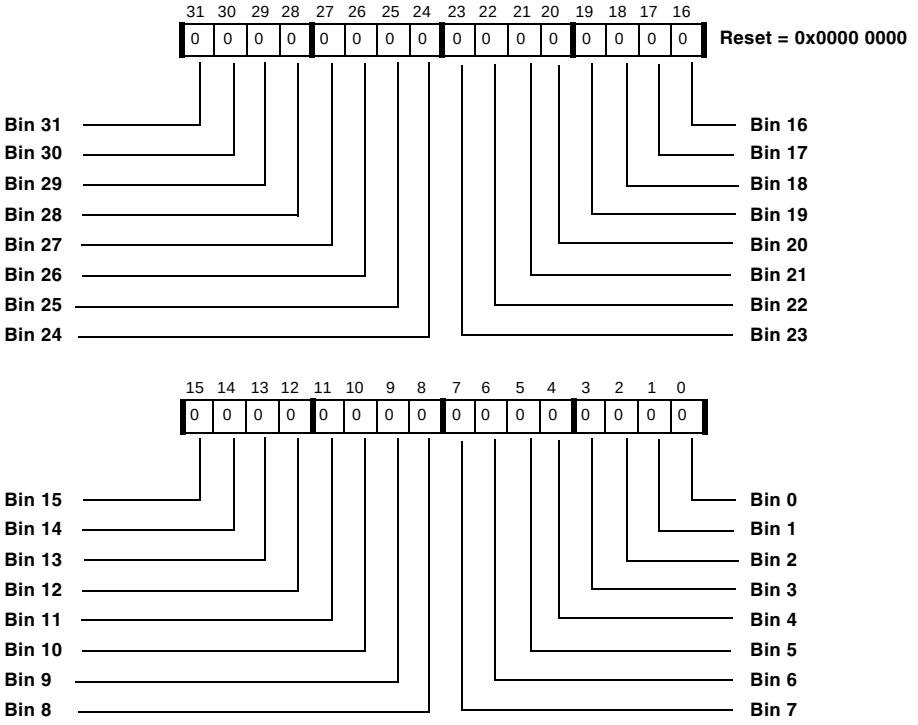


Figure 21-16. EMAC_HASHLO Register

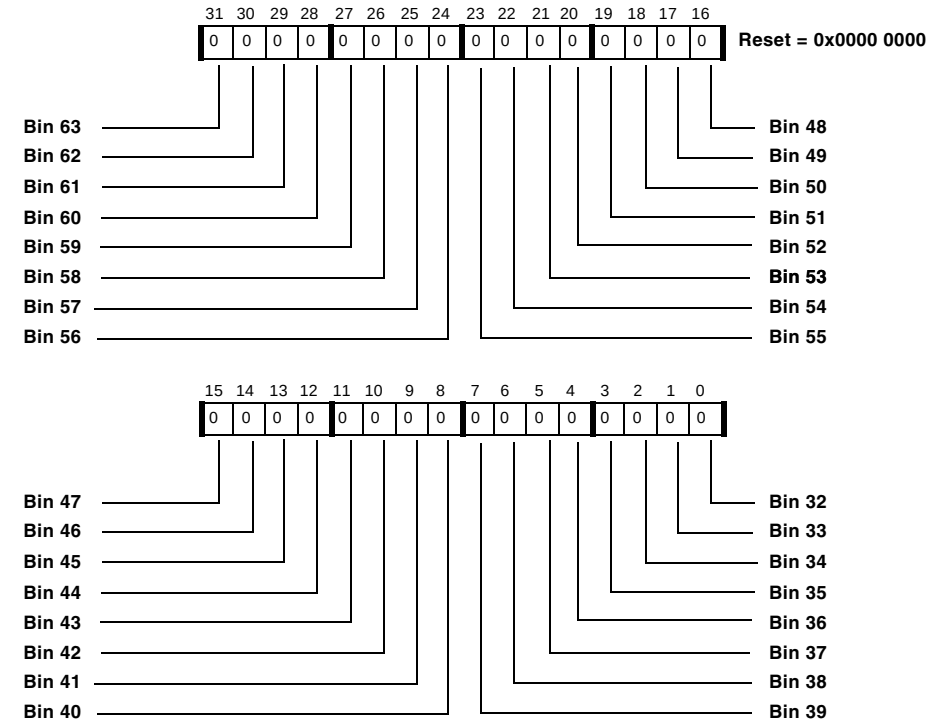


Figure 21-17. EMAC_HASHHI Register

MAC Station Management Address (EMAC_STAADD) Register

The EMAC_STAADD register, shown in Figure 21-18, controls the transactions between the MII management (MIM) block and the registers on the

external PHY. These transactions are used to appropriately configure the PHY and monitor its performance.

MAC Station Management Address Register (EMAC_STAADD)

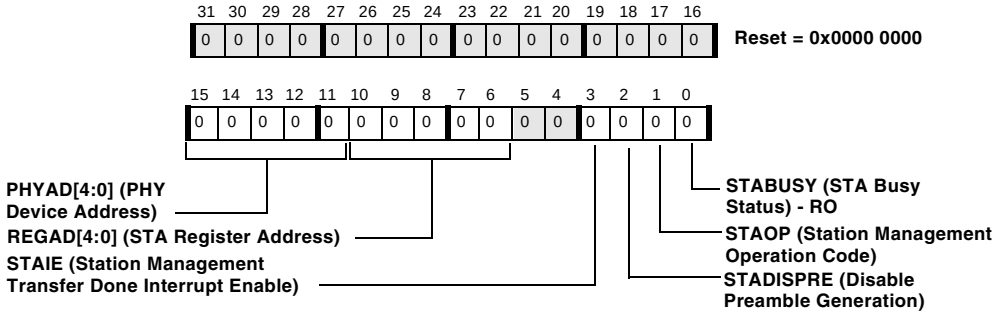

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
PHYAD[4:0] (PHY Device Address)
REGAD[4:0] (STA Register Address)
STAIE (Station Management Transfer Done Interrupt Enable)
STABUSY (STA Busy Status) - RO
STAOP (Station Management Operation Code)
STADISPRE (Disable Preamble Generation)

Figure 21-18. EMAC_STAADD Register

Additional information for the EMAC_STAADD register bits includes:

- **Station management transfer done interrupt enable** (STAIE)

  0 – Interrupt not enabled.

  1 – Enables an Ethernet event interrupt at the completion of a station management register access
  (when STABUSY changes from 1 to 0).

- **Disable preamble generation** (STADISPRE)

  0 – Preamble generation for station management transfers not disabled.

  1 – Preamble generation (32 ones) for station management transfers disabled.

- **Station management operation code** (STAOP)

  0 – Read

1 – Write

- **STA busy status** (STABUSY)

  This bit should be set by the application software in order to initiate a station management register access. This bit is automatically cleared when the access is complete. The MAC ignores new transfer requests made while the serial interface is busy. Writes to the STA address or data registers are discarded if STABUSY is 1.

  0 – No operation.

  1 – Initiate a station management register access across MDC/MDIO.

MAC Station Management Data (EMAC_STADAT) Register

The EMAC_STADAT register, shown in Figure 21-19, contains either the data to be written to the PHY register specified in the EMAC_STAADD register, or the data read from the PHY register whose address is specified in the EMAC_STAADD register.

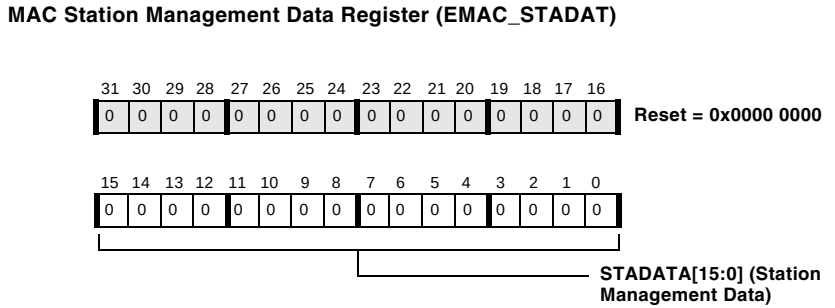


Figure 21-19. EMAC_STADAT Register

MAC Flow Control (EMAC_FLC) Register

The EMAC_FLC register, shown in Figure 21-20, controls the generation and reception of control frames by the MAC. The control frame fields are selected as specified in the IEEE 802.3 specification. When flow control is enabled, the MAC acts upon MAC control pause frames received without errors. When an error-free MAC control pause frame is received (with length/type = MacControl = 88-08 and with opcode = pause = 00-01), the transmitter defers starting new frames for the number of slot times specified by the pause time field in the control frame.

The MAC can also generate and transmit a MAC control pause frame when the EMAC_FLC register is written with FLCBUSY = 1 and FLCPAUSE equal to the number of slot times of deferral being requested.

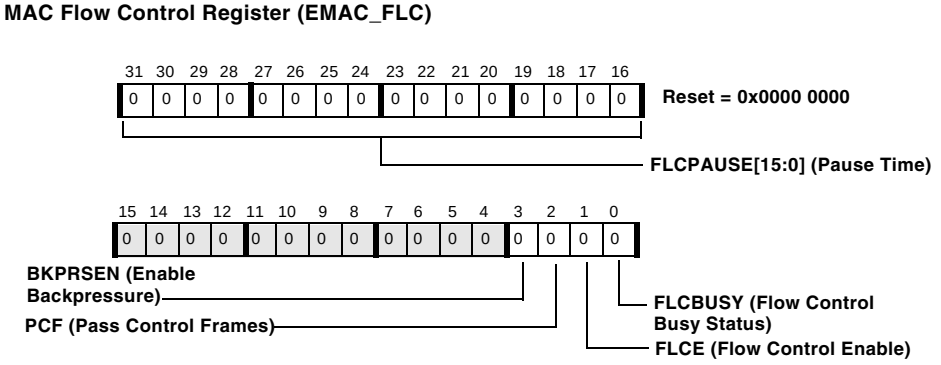

MAC Flow Control Register (EMAC_FLC)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
FLCPAUSE[15:0] (Pause Time)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
BKPRSEN (Enable Backpressure)
PCF (Pass Control Frames)
FLCBUSY (Flow Control Busy Status)
FLCE (Flow Control Enable)

Figure 21-20. EMAC_FLC Register

Additional information for the EMAC_FLC register bits includes:

- **Pause time** (FLCPAUSE)

  The number of slot times for which the transmission of new frames is deferred.

- **Enable back pressure** (BKPRSEN)

Available only in half-duplex mode, this bit can be used as a form of flow control.

0 – Transmit and receive function is normal.

1 – Prevents frame reception by colliding with (continuously transmitting a jam pattern during) every incoming frame.

- **Pass control frames** (PCF)

  When cleared, the PCF bit causes the frame filter to reject all control frames (frames with length/type field equal to 88-08). When cleared, error-free pause control frames are still interpreted (if enabled by FLCE) but are not delivered via DMA.

  0 – Do not pass control frames.

  1 – Pass control frames.

- **Flow control enable** (FLCE)

  When set, this bit enables interpretation of MAC control pause frames that are received without errors.

  0 – Flow control not enabled.

  1 – Flow control enabled.

- **FLC busy status** (FLCBUSY)

  Setting this bit triggers the MAC to send a control frame. The MAC automatically clears the FLCBUSY bit once the control frame has been transferred onto the physical medium. Writes to the EMAC_FLC register are discarded if FLCBUSY is 1.

  0 – No operation.

  1 – Initiate sending flow control frame.

MAC VLAN1 Tag (EMAC_VLAN1) and MAC VLAN2 Tag (EMAC_VLAN2)Registers

The EMAC_VLAN1 register, shown in Figure 21-21, and the EMAC_VLAN2 register, shown in Figure 21-22, contain the tag fields used to identify VLAN frames. The MAC compares the 13th and 14th bytes of the incoming frame field to the values contained in these registers, so that the 13th frame byte is compared to the most significant byte of the registers and the 14th frame byte is compared to the least significant byte of the registers. If a match is found, the appropriate bit is set in the EMAC_RX_STAT register. The legal length of the frame is then increased from 1518 bytes to either 1522 bytes in the case of a VLAN1 match or 1538 bytes for a VLAN2 match.

MAC VLAN1 Tag Register (EMAC_VLAN1)

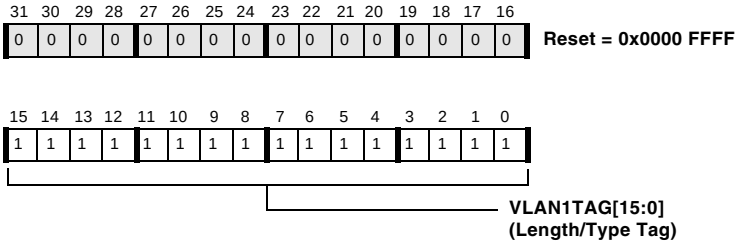


Figure 21-21. EMAC_VLAN1 Register

MAC VLAN2 Tag Register (EMAC_VLAN2)

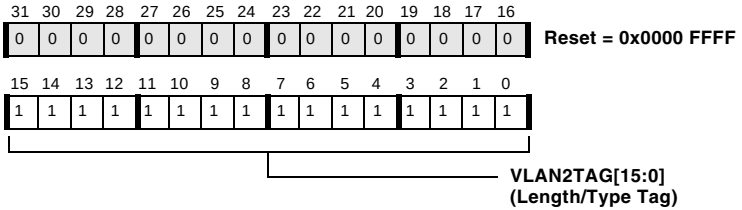


Figure 21-22. EMAC_VLAN2 Register

MAC Wakeup Frame Control and Status (EMAC_WKUP_CTL) Register

The EMAC_WKUP_CTL register, shown in Figure 21-23, contains data pertaining to the MAC's remote wakeup status and capabilities. A write to the EMAC_WKUP_CTL register causes an update into the receive clock domain of all the wakeup filter registers. Changes to these other registers do not affect the operation of the MAC until the EMAC_WKUP_CTL register is written. For this reason, it is recommended that the wakeup filters be

programmed by writing all of the other registers first, and writing the EMAC_WKUP_CTL register last.

MAC Wakeup Frame Control and Status Register (EMAC_WKUP_CTL)

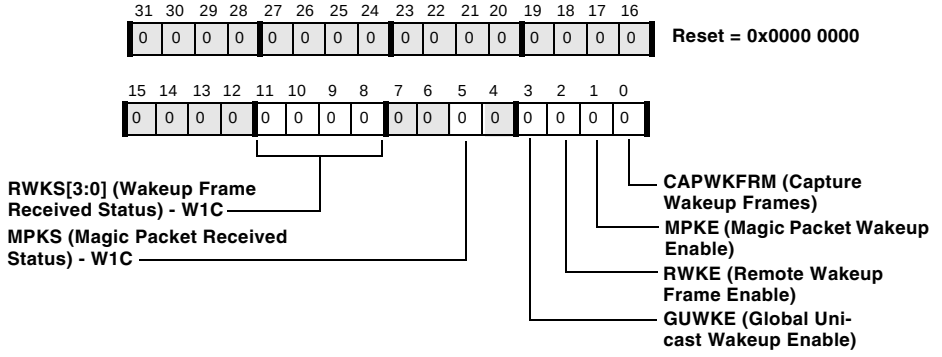

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RWKS[3:0] (Wakeup Frame Received Status) - W1C
MPKS (Magic Packet Received Status) - W1C
CAPWKFRM (Capture Wakeup Frames)
MPKE (Magic Packet Wakeup Enable)
RWKE (Remote Wakeup Frame Enable)
GUWKE (Global Unicast Wakeup Enable)

Figure 21-23. EMAC_WKUP_CTL Register

Additional information for the EMAC_WKUP_CTL register bits includes:

- **Wakeup frame received status** (RWKS)

  These four frame status bits flag the receipt of wakeup frames corresponding to the respective wakeup frame filters.

- **Magic packet received status** (MPKS)

  This bit is set by the MAC when it receives the magic packet received wakeup call. The MAC then resumes operation in the normal powered-up mode.

  0 – Magic packet not received.

  1 – Magic packet received.

- **Global unicast wake enable** (GUWKE)

When set, configures the MAC to wake up from the power-down mode on receipt of a global unicast frame. Such a frame has the MAC address [1:0] bits cleared.

0 – Global unicast wake not enabled.

1 – Global unicast wake enabled.

- **Remote wakeup frame enable** (RWKE)

  When set, this bit enables the remote wakeup frame power-down mode.

  0 – Remote wakeup frame not enabled.

  1 – Remote wakeup frame enabled.

- **Magic packet wakeup enable** (MPKE)

  When set, this bit enables the magic packet wakeup power-down mode.

  0 – Magic packet wakeup not enabled.

  1 – Magic packet wakeup enabled.

- **Capture wakeup frames** (CAPWKFRM)

  0 – The RX DMA pathway is disabled when MPKE or RWKE is set.

  1 – RX frames are delivered via DMA while in power-down mode (when either MPKE or RWKE is set).

MAC Wakeup Frame0 Byte Mask (EMAC_WKUP_FFMSK0)
MAC Wakeup Frame1 Byte Mask (EMAC_WKUP_FFMSK1)
MAC Wakeup Frame2 Byte Mask (EMAC_WKUP_FFMSK2)
MAC Wakeup Frame3 Byte Mask (EMAC_WKUP_FFMSK3) Registers

The EMAC_WKUP_FFMSK0, EMAC_WKUP_FFMSK1, EMAC_WKUP_FFMSK2, and EMAC_WKUP_FFMSK3 registers (see Figure 21-24 through Figure 21-27) are a part of the mechanism used to select which bytes in a received frame are used for CRC computation. Each bit in these registers functions as a byte enable. If a bit i is set, then the byte (offset + i) is used for CRC computation, where offset is contained in the EMAC_WKUP_FFOFF register.

For example, to identify a wakeup packet containing the byte sequence (0x80, 0x81, 0x82) in bytes 14, 15, and 17, the EMAC_WKUP_FFOFF register should be set to 14 and the byte mask should be set to 0x000B. This byte mask has bits 0, 1, and 3 set, so that bytes 14+0, 14+1, and 14+3 are selected.

MAC Wakeup Frame0 Byte Mask Register (EMAC_WKUP_FFMSK0)

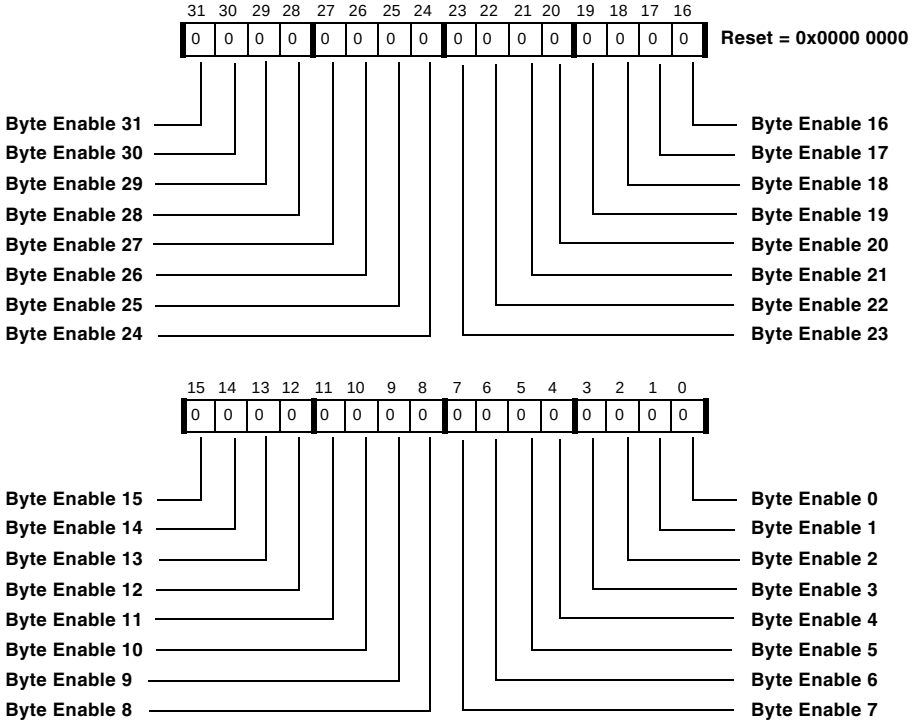

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
Byte Enable 31
Byte Enable 30
Byte Enable 29
Byte Enable 28
Byte Enable 27
Byte Enable 26
Byte Enable 25
Byte Enable 24
Byte Enable 16
Byte Enable 17
Byte Enable 18
Byte Enable 19
Byte Enable 20
Byte Enable 21
Byte Enable 22
Byte Enable 23
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Byte Enable 15
Byte Enable 14
Byte Enable 13
Byte Enable 12
Byte Enable 11
Byte Enable 10
Byte Enable 9
Byte Enable 8
Byte Enable 0
Byte Enable 1
Byte Enable 2
Byte Enable 3
Byte Enable 4
Byte Enable 5
Byte Enable 6
Byte Enable 7

Figure 21-24. EMAC_WKUP_FFMSK0 Register

MAC Wakeup Frame1 Byte Mask Register (EMAC_WKUP_FFMSK1)

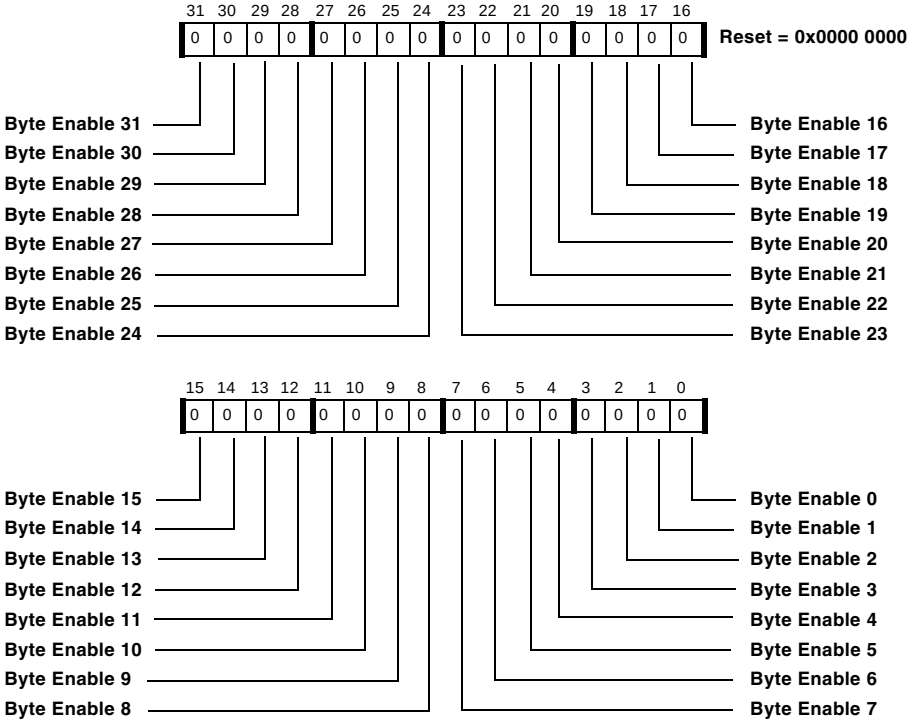

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
Byte Enable 31
Byte Enable 30
Byte Enable 29
Byte Enable 28
Byte Enable 27
Byte Enable 26
Byte Enable 25
Byte Enable 24
Byte Enable 16
Byte Enable 17
Byte Enable 18
Byte Enable 19
Byte Enable 20
Byte Enable 21
Byte Enable 22
Byte Enable 23
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Byte Enable 15
Byte Enable 14
Byte Enable 13
Byte Enable 12
Byte Enable 11
Byte Enable 10
Byte Enable 9
Byte Enable 8
Byte Enable 0
Byte Enable 1
Byte Enable 2
Byte Enable 3
Byte Enable 4
Byte Enable 5
Byte Enable 6
Byte Enable 7

Figure 21-25. EMAC_WKUP_FFMSK1 Register

MAC Wakeup Frame2 Byte Mask Register (EMAC_WKUP_FFMSK2)

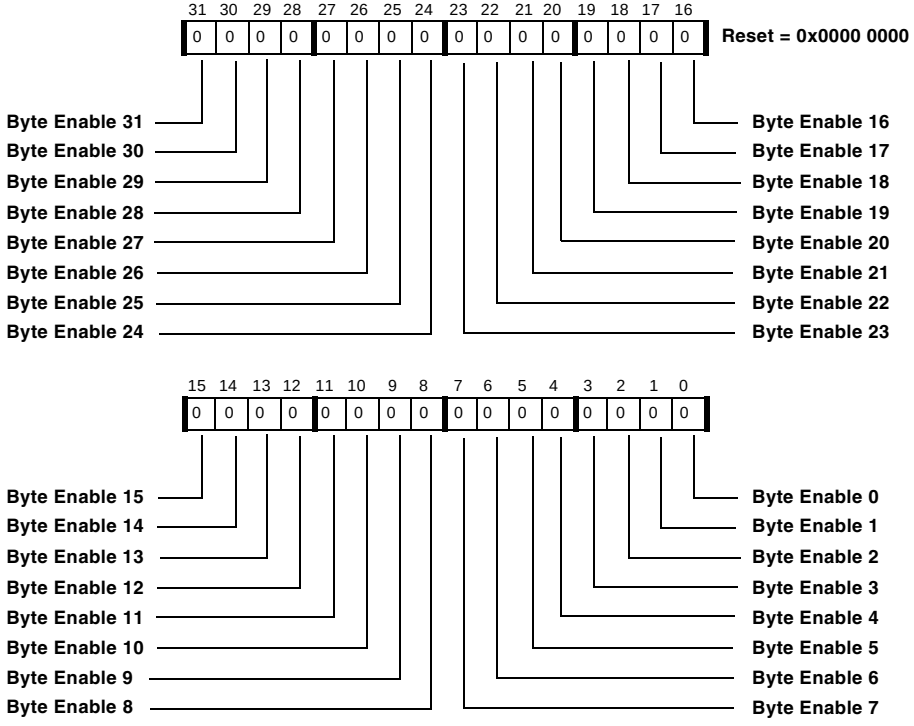

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
Byte Enable 31
Byte Enable 30
Byte Enable 29
Byte Enable 28
Byte Enable 27
Byte Enable 26
Byte Enable 25
Byte Enable 24
Byte Enable 16
Byte Enable 17
Byte Enable 18
Byte Enable 19
Byte Enable 20
Byte Enable 21
Byte Enable 22
Byte Enable 23
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Byte Enable 15
Byte Enable 14
Byte Enable 13
Byte Enable 12
Byte Enable 11
Byte Enable 10
Byte Enable 9
Byte Enable 8
Byte Enable 0
Byte Enable 1
Byte Enable 2
Byte Enable 3
Byte Enable 4
Byte Enable 5
Byte Enable 6
Byte Enable 7

Figure 21-26. EMAC_WKUP_FFMSK2 Register

MAC Wakeup Frame3 Byte Mask Register (EMAC_WKUP_FFMSK3)

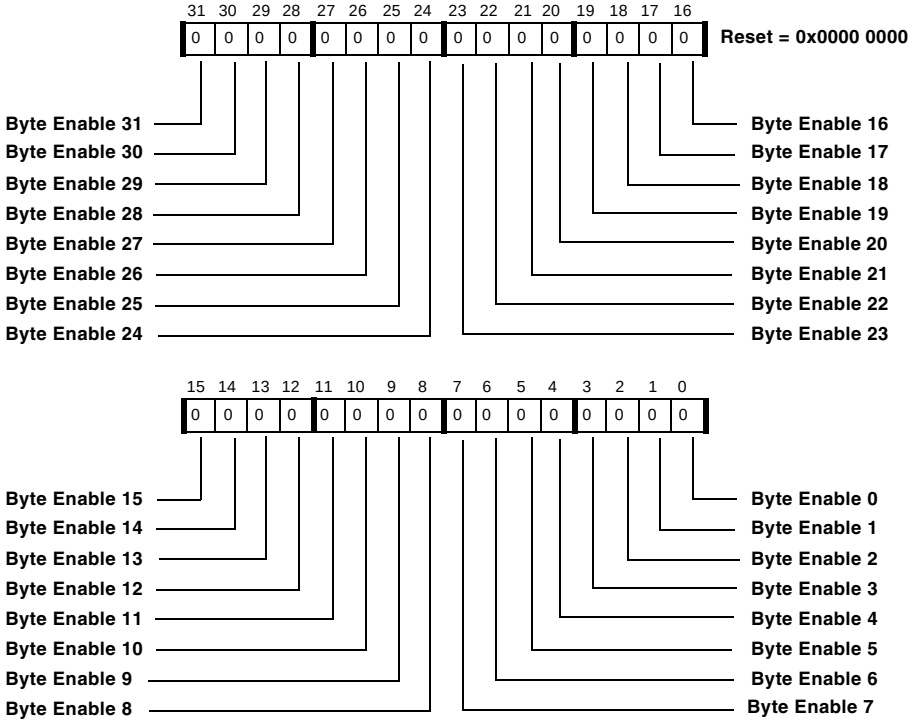

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
Byte Enable 31
Byte Enable 30
Byte Enable 29
Byte Enable 28
Byte Enable 27
Byte Enable 26
Byte Enable 25
Byte Enable 24
Byte Enable 16
Byte Enable 17
Byte Enable 18
Byte Enable 19
Byte Enable 20
Byte Enable 21
Byte Enable 22
Byte Enable 23
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Byte Enable 15
Byte Enable 14
Byte Enable 13
Byte Enable 12
Byte Enable 11
Byte Enable 10
Byte Enable 9
Byte Enable 8
Byte Enable 0
Byte Enable 1
Byte Enable 2
Byte Enable 3
Byte Enable 4
Byte Enable 5
Byte Enable 6
Byte Enable 7

Figure 21-27. EMAC_WKUP_FFMSK3 Register

MAC Wakeup Frame Filter Commands (EMAC_WKUP_FFCMD) Register

The EMAC_WKUP_FFCMD register, shown in Figure 21-28, regulates which of the four frame filter registers are enabled and if so, whether they are configured for unicast or multicast address filtering.

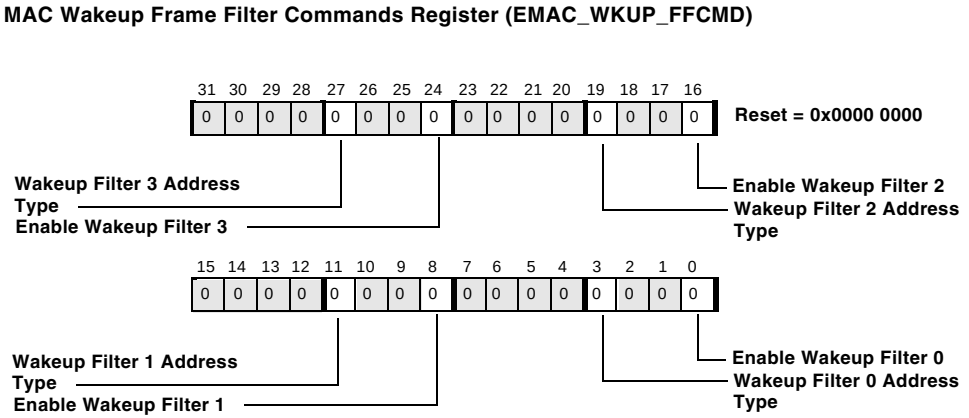

MAC Wakeup Frame Filter Commands Register (EMAC_WKUP_FFCMD)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
Wakeup Filter 3 Address Type
Enable Wakeup Filter 3
Enable Wakeup Filter 2
Wakeup Filter 2 Address Type
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Wakeup Filter 1 Address Type
Enable Wakeup Filter 1
Enable Wakeup Filter 0
Wakeup Filter 0 Address Type

Figure 21-28. EMAC_WKUP_FFCMD Register

Additional information for the EMAC_WKUP_FFCMD register bits includes:

- **Wakeup filter 3 address type**

  0 – Unicast

  1 – Multicast

- **Enable wakeup filter 3**

  0 – Wakeup filter 3 not enabled.

  1 – Wakeup filter 3 enabled.

- **Wakeup filter 2 address type**

  0 – Unicast

1 – Multicast

- **Enable wakeup filter 2**

  0 – Wakeup filter 2 not enabled.

  1 – Wakeup filter 2 enabled.

- **Wakeup filter 1 address type**

  0 – Unicast

  1 – Multicast

- **Enable wakeup filter 1**

  0 – Wakeup filter 1 not enabled.

  1 – Wakeup filter 1 enabled.

- **Wakeup filter 0 address type**

  0 – Unicast

  1 – Multicast

- **Enable wakeup filter 0**

  0 – Wakeup filter 0 not enabled.

  1 – Wakeup filter 0 enabled.

Ethernet MAC Wakeup Frame Filter Offsets (EMAC_WKUP_FFOFF) Register

The EMAC_WKUP_FFOFF register, shown in Figure 21-29, contains the byte offsets for CRC computation to be performed on potential wakeup frames.

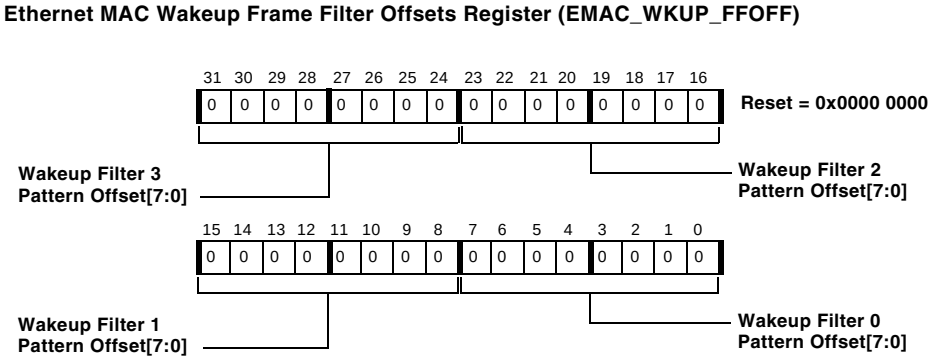

Ethernet MAC Wakeup Frame Filter Offsets Register (EMAC_WKUP_FFOFF)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
Wakeup Filter 3 Pattern Offset[7:0]
Wakeup Filter 2 Pattern Offset[7:0]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Wakeup Filter 1 Pattern Offset[7:0]
Wakeup Filter 0 Pattern Offset[7:0]

Figure 21-29. EMAC_WKUP_FFOFF Register

MAC Wakeup Frame Filter CRC0/1 (EMAC_WKUP_FFCRC0) and CRC2/3 (EMAC_WKUP_FFCRC1) Registers

The EMAC_WKUP_FFCRC0 register, shown in Figure 21-30, and the EMAC_WKUP_FFCRC1 register, shown in Figure 21-31, should be loaded with

the results of the CRC computations for the relevant wakeup frame bytes. See “Remote Wake-up Filters” on page 21-34.

MAC Wakeup Frame Filter CRC0/1 Register (EMAC_WKUP_FFCRC0)

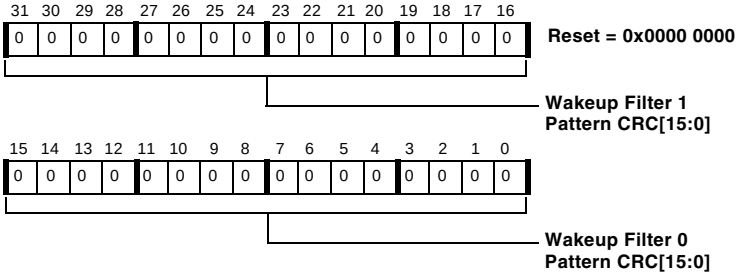


Figure 21-30. EMAC_WKUP_FFCRC0 Register

MAC Wakeup Frame Filter CRC2/3 Register (EMAC_WKUP_FFCRC1)

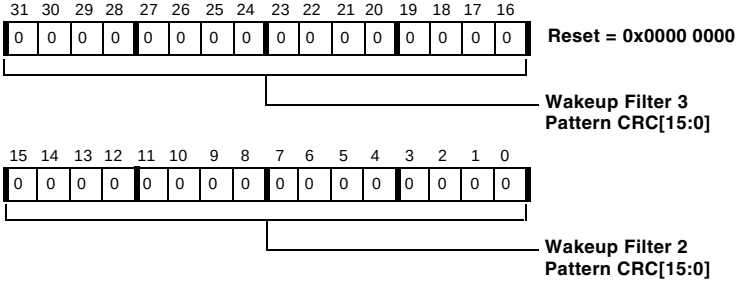


Figure 21-31. EMAC_WKUP_FFCRC1 Register

System Interface Register Group

The SIF block registers control and monitor the MAC’s interactions with the Blackfin processor peripheral subsystem and the external PHY. The SIF block has several frame status registers whose bit descriptions can be found in “Ethernet MAC Frame Status Registers” on page 21-94.

MAC System Control (EMAC_SYSCTL) Register

The EMAC_SYSCTL register, shown in Figure 21-32, is used to set up MAC controls.

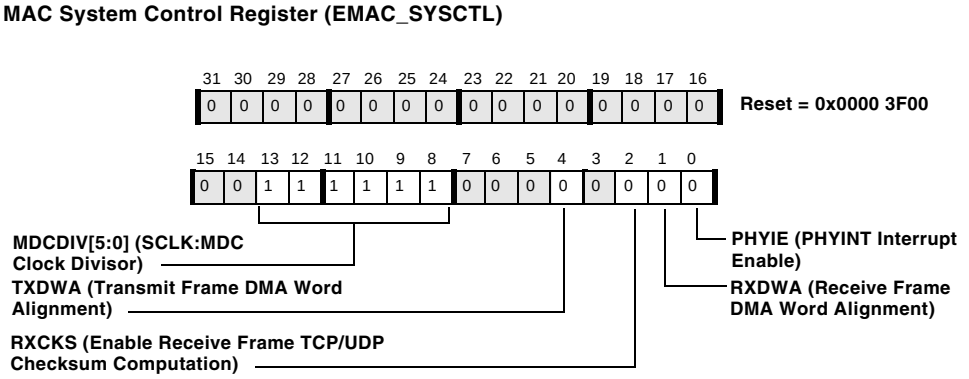

MAC System Control Register (EMAC_SYSCTL)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 3F00
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0
MDCDIV[5:0] (SCLK:MDC Clock Divisor)
TXDWA (Transmit Frame DMA Word Alignment)
RXCKS (Enable Receive Frame TCP/UDP Checksum Computation)
PHYIE (PHYINT Interrupt Enable)
RXDWA (Receive Frame DMA Word Alignment)

Figure 21-32. EMAC_SYSCTL Register

Additional information for the EMAC_SYSCTL register bits includes:

- **SCLK:MDC clock divisor** (MDCDIV[5:0])

  This field contains the clock divisor that determines the ratio between the Blackfin system clock (SCLK) and the MAC data clock (MDC). The 6-bit ratio N determines the MDC rate as:

  MDC = SCLK/(2 x (N + 1)).

- **Transmit frame DMA word alignment** (TXDWA)

  This bit determines whether outgoing frame data is aligned on odd or even 16-bit boundaries in memory.

  0 – Odd word alignment.

  1 – Even word alignment.

- **Enable receive frame TCP/UDP checksum computation** (RXCKS)

0 – Receive frame TCP/UDP checksum computation not enabled.

1 – TCP/UDP checksum computation on received frames enabled.

- **Receive frame DMA word alignment** (RXDWA)

  This bit determines whether incoming frames are aligned on odd or even 16-bit boundaries in memory.

  0 – Even word alignment.

  1 – Odd word alignment.

- **PHYINT interrupt enable** (PHYIE)

  0 – PHYINT interrupt not enabled.

  1 – PHYINT interrupt enabled.

MAC System Status (EMAC_SYSTAT) Register

The EMAC_SYSTAT register, shown in Figure 21-33, contains a range of interrupt status bits that signal the occurrence of significant Ethernet

events to the application. Detailed descriptions of the functionality can be found in the section entitled “Ethernet Event Interrupts” on page 21-39.

MAC System Status Register (EMAC_SYSTAT)

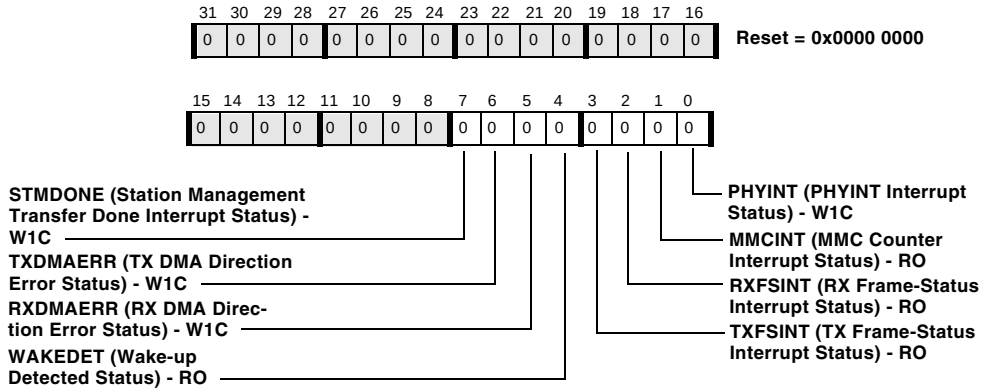

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
STMDONE (Station Management Transfer Done Interrupt Status) - W1C
TXDMAERR (TX DMA Direction Error Status) - W1C
RXDMAERR (RX DMA Direction Error Status) - W1C
WAKEDET (Wake-up Detected Status) - RO
PHYINT (PHYINT Interrupt Status) - W1C
MMCINT (MMC Counter Interrupt Status) - RO
RXFSINT (RX Frame-Status Interrupt Status) - RO
TXFSINT (TX Frame-Status Interrupt Status) - RO

Figure 21-33. EMAC_SYSTAT Register

Additional information for the EMAC_SYSTAT register bits includes:

- **Station management transfer done interrupt status** (STMDONE)

  This bit is set when a station management transfer on MDC/MDIO has completed, provided the STAIE interrupt enable control bit is set in the EMAC_STAADD register.

- **TX DMA direction error status** (TXDMAERR)

  This bit is set if a TX data or status DMA request is granted by the DMA channel with transfer in the wrong direction. Data should be memory-read, status should be memory-write. This interrupt is non-maskable in the Ethernet MAC.

- **RX DMA direction error status** (RXDMAERR)

This bit it set if an RX data or status DMA request is granted by the DMA channel with transfer in the wrong (memory-read) direction. This interrupt is non-maskable in the Ethernet MAC.

- **Wakeup detected status** (WAKEDET)

  To clear this bit, write 1 to the EMAC_WKUP_CTL register.

  0 – Wakeup not detected.

  1 – Wakeup detected.

- **TX frame-status interrupt status** (TXFSINT)

  To clear this bit, write 1s to the EMAC_RX_STKY register bits.

  0 – TX frame-status interrupt has not occurred.

  1 – TX frame-status interrupt has occurred.

- **RX frame-status interrupt status** (RXFSINT)

  To clear this bit, write 1s to the EMAC_RX_STKY register bits.

  0 – RX frame-status interrupt has not occurred.

  1 – RX frame-status interrupt has occurred.

- **MMC counter interrupt status** (MMCINT)

  To clear this bit, write 1 to the EMAC_MMC_RIRQS or EMAC_MMC_TIRQS register.

  0 – MMC counter interrupt has not occurred.

  1 – MMC counter interrupt has occurred.

- **PHYINT interrupt status** (PHYINT)

  0 – PHYINT interrupt has not occurred.

1 – PHYINT interrupt has occurred.

Ethernet MAC Frame Status Registers

The Ethernet MAC frame status registers keep track of the status of each frame received or transmitted, as well as the status of MMC interrupts.

Ethernet MAC RX Current Frame Status (EMAC_RX_STAT) Register

The EMAC_RX_STAT register, shown in Figure 21-34, tells the status of the most recently completed receive frame, including type of error for cases where an error occurs. When the receive complete bit is set, exactly one of bits 13 through 20 is 1. Bits 13 through 20 indicate the receive status as defined in IEEE 802.3, section 4.3.2. In case of multiple errors, errors are prioritized in the order listed in Table 21-4 on page 21-20. Bits 18 and 19 identify frames which are not considered received by the station and also are not considered errors. (See section 4.1.2.1.2 and section 4.2.4.2.2 of IEEE 802.3.) Bit 20 identifies frames damaged within the MAC sublayer.

Note if the PB (pass bad frames) bit is 0, then delivery via DMA of frames with status bits 14 through 18 or 20 is cancelled. The DMA buffer is

reused for the next frame. If the PR (promiscuous) bit is 0, then frames with bit 19 set are not delivered (the DMA is never initiated).

Ethernet MAC RX Current Frame Status Register (EMAC_RX_STAT)
All bits in this register are RO.

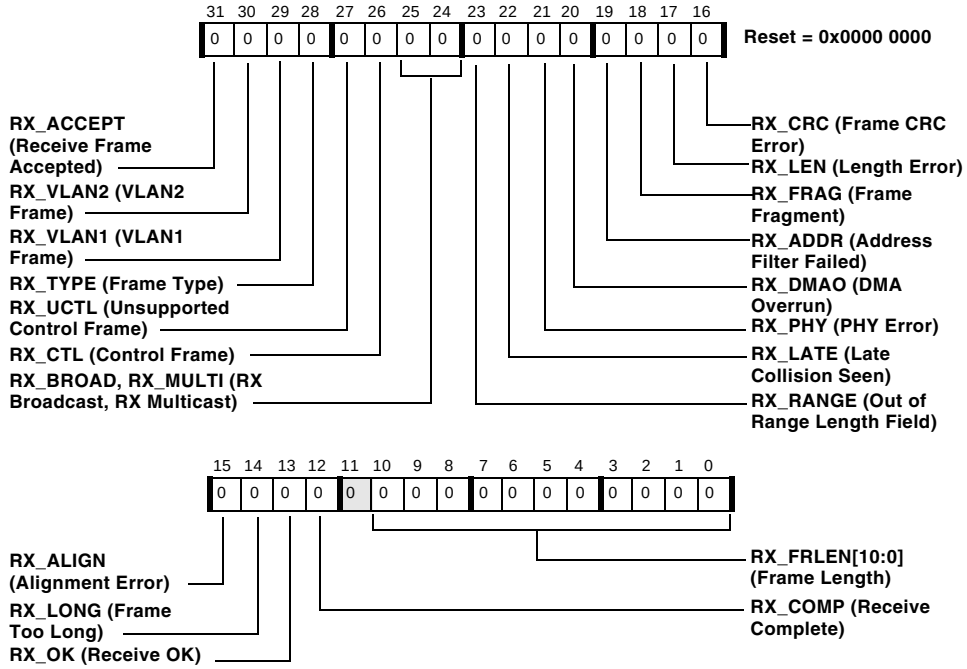

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
RX_ACCEPT (Receive Frame Accepted)
RX_VLAN2 (VLAN2 Frame)
RX_VLAN1 (VLAN1 Frame)
RX_TYPE (Frame Type)
RX_UCTL (Unsupported Control Frame)
RX_CTL (Control Frame)
RX_BROAD, RX_MULTI (RX Broadcast, RX Multicast)
RX_CRC (Frame CRC Error)
RX_LEN (Length Error)
RX_FRAG (Frame Fragment)
RX_ADDR (Address Filter Failed)
RX_DMAO (DMA Overrun)
RX_PHY (PHY Error)
RX_LATE (Late Collision Seen)
RX_RANGE (Out of Range Length Field)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RX_ALIGN (Alignment Error)
RX_LONG (Frame Too Long)
RX_OK (Receive OK)
RX_FRLEN[10:0] (Frame Length)
RX_COMP (Receive Complete)

Figure 21-34. EMAC_RX_STAT Register

Additional information for the EMAC_RX_STAT register bits includes:

- **Receive frame accepted** (RX_ACCEPT)

  0 – Receive frame not accepted.

1 – The receive frame was accepted, based on the address filter result and the frame filtering modes in the EMAC_OPMODE register. Note that this does not imply a status of receiveOK. If the RA (receive all) control bit is 0, then the only frames delivered by DMA are the frames whose receive frame accepted status bit is 1.

- **VLAN2 frame** (RX_VLAN2)

  0 – The frame does not meet those conditions.

  1 – The frame is a valid tagged frame with a length/type field matching the EMAC_VLAN2 register, and with status of receiveOK.

- **VLAN1 frame** (RX_VLAN1)

  0 – The frame does not meet those conditions.

  1 – The frame is a valid tagged frame with a length/type field matching the EMAC_VLAN1 register, and with status of receiveOK.

- **Frame type** (RX_TYPE)

  0 – The frame is not of that type.

  1 – The frame is a valid typed frame, with status of receiveOK and with a length/type field greater than or equal to 0x600.

- **Unsupported control frame** (RX_UCTL)

  0 – The frame does not meet those conditions.

  1 – The frame is a valid MAC control frame (with status of receiveOK and with a length/type field equal to 802.3_MAC_Control, 88-08), but does not contain the pause opcode, or is not 64 bits in length, or is received in half-duplex mode.

- **Control frame** (RX_CTL)

1 – The frame is a valid MAC control frame in full duplex mode with status of receiveOK, with a length/type field equal to MAC_Control, 88-08, with length of 64 bytes, and with a MAC control opcode field equal to the pause opcode (00-01).

0 – The frame does not meet those conditions.

- **RX broadcast, RX multicast** (RX_BROAD, RX_MULTI)

  00 – Unicast address

  01 – Group address

  10 – Broadcast address

  11 – Illegal

- **Out of range length field** (RX_RANGE)

  0 – The frame's length was not out of range.

  1 – The frame's length/type field was consistent with the length interpretation (<1536 = 0x600) but was greater than the maximum allowable frame size in bytes, as indicated by the frame too long bit).

- **Late collision seen** (RX_LATE)

  0 – Late collision not detected.

  1 – A collision was detected after the first 64 bytes of the packet.

- **PHY error** (RX_PHY)

  0 – No PHY error.

  1 – RX_ER was asserted at some time during the frame. This condition always causes the FCS check to fail.

- **DMA overrun** (RX_DMAO)

0 – No DMA overrun.

1 – The received frame was truncated due to failure of the FIFO/DMA channel to continuously store data during DMA transfer to memory.

- **Address filter failed** (RX_ADDR)

  0 – Address did not fail.

  1 – The destination address did not pass the address filters specified by the station MAC address, the EMAC_HASHLO/EMAC_HASHHI registers, and the filter modes in the EMAC_OPMODE register.

- **Frame fragment** (RX_FRAG)

  0 – Frame length was at least 64 bytes.

  1 – Frame length was less than the minimum frame size (64 bytes).

- **Length error** (RX_LEN)

  0 – No frame length error.

  1 – The frame's length/type field does not match the length of received data and is consistent with the length interpretation (< 0x600), although the frame had no "frame too long" errors and had a valid FCS.

- **Frame CRC error** (RX_CRC)

  0 – No frame CRC error.

  1 – The frame failed FCS validation, but had neither a "frame too long" error nor a partial number of octets. Note if RX_ER is asserted by the PHY during frame reception, the FCS validation will fail.

- **Alignment error** (RX_ALIGN)

  0 – No alignment error.

1 – The frame ended with a partial octet and failed RCS validation, but had no frame too long error.

- **Frame too long** (RX_LONG)

  0 – Frame is not too long.

  1 – The number of octets received is greater than the maximum Ethernet frame size. Maximum frame size is 1522 bytes for a frame whose length/type field matches the EMAC_VLAN1 register, 1538 bytes for a frame whose length/type field matches the EMAC_VLAN2 register, or 1518 for all other frames. The frame data delivered by DMA is truncated to 1556 (0x614) bytes in all cases.

- **Receive OK** (RX_OK)

  0 – A receive error occurred.

  1 – There was no receive error.

- **Receive complete** (RX_COMP)

  This bit is cleared on reset and when the MAC RX is enabled (RE changes from 0 to 1). Frames that fail the address filter or the frame filter are not delivered by DMA, unless overridden by the RA (receive all) control bit. Note that in the RX frame status buffer written to memory by DMA, the receive complete bit is always 1. This bit acts as a semaphore, indicating that DMA of the frame has completed.

  0 – The first RX frame is not yet complete.

  1 – The first RX frame is complete.

- **Frame length** (RX_FRLEN)

  The number of bytes in the frame. If the ASTP bit is set, the pad and FCS are not included in the length.

Ethernet MAC RX Sticky Frame Status (EMAC_RX_STKY) Register

The EMAC_RX_STKY register, shown in Figure 21-35, accumulates state across multiple frames, unless software clears it after every frame.

Ethernet MAC RX Sticky Frame Status Register (EMAC_RX_STKY)
All bits in this register are W1C.

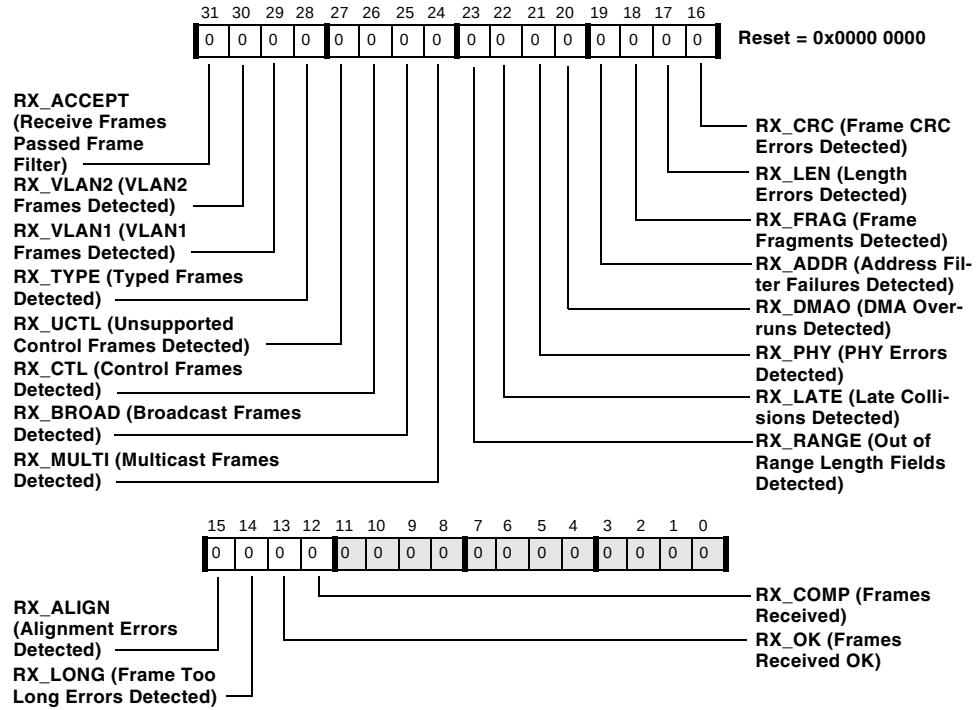

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
RX_ACCEPT (Receive Frames Passed Frame Filter)
RX_VLAN2 (VLAN2 Frames Detected)
RX_VLAN1 (VLAN1 Frames Detected)
RX_TYPE (Typed Frames Detected)
RX_UCTL (Unsupported Control Frames Detected)
RX_CTL (Control Frames Detected)
RX_BROAD (Broadcast Frames Detected)
RX_MULTI (Multicast Frames Detected)
RX_CRC (Frame CRC Errors Detected)
RX_LEN (Length Errors Detected)
RX_FRAG (Frame Fragments Detected)
RX_ADDR (Address Filter Failures Detected)
RX_DMAO (DMA Overruns Detected)
RX_PHY (PHY Errors Detected)
RX_LATE (Late Collisions Detected)
RX_RANGE (Out of Range Length Fields Detected)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RX_ALIGN (Alignment Errors Detected)
RX_LONG (Frame Too Long Errors Detected)
RX_COMP (Frames Received)
RX_OK (Frames Received OK)

Figure 21-35. EMAC_RX_STKY Register

Additional information for the EMAC_RX_STKY register bits includes:

- **Receive frames passed frame filter** (RX_ACCEPT)

  0 – No receive frames passed the frame filter.

1 – At least one receive frame passed the frame filter.

- **VLAN2 frames detected** (RX_VLAN2)

  0 – No VLAN2 frames were detected.

  1 – At least one VLAN2 frame was detected.

- **VLAN1 frames detected** (RX_VLAN1)

  0 – No VLAN1 frames were detected.

  1 – At least one VLAN1 frame was detected.

- **Typed frames detected** (RX_TYPE)

  0 – No typed frames were detected.

  1 – At least one typed frame was detected.

- **Unsupported control frames detected** (RX_UCTL)

  0 – No unsupported control frames were detected.

  1 – At least one unsupported control frame was detected.

- **Control frames detected** (RX_CTL)

  0 – No control frames were detected.

  1 – At least one control frame was detected.

- **Broadcast frames detected** (RX_BROAD)

  0 – No broadcast frames were detected.

  1 – At least one broadcast frame was detected.

- **Multicast frames detected** (RX_MULTI)

  0 – No multicast frames were detected.

1 – At least one multicast frame was detected.

- **Out of range length fields detected** (RX_RANGE)

  0 – No out of range length fields were detected.

  1 – At least one out of range length field was detected.

- **Late collisions detected** (RX_LATE)

  0 – No late collisions were detected.

  1 – At least one collision was detected after the first 64 bytes of the packet.

- **PHY errors detected** (RX_PHY)

  0 – No PHY errors were detected.

  1 – At least one PHY error was detected.

- **DMA overruns detected** (RX_DMAO)

  0 – No DMA overruns were detected.

  1 – At least one DMA overrun was detected.

- **Address filter failures detected** (RX_ADDR)

  0 – No address filter failures were detected.

  1 – At least one address filter failure was detected.

- **Frame fragments detected** (RX_FRAG)

  0 – No frame fragments were detected.

  1 – At least one frame fragment was detected.

- **Length errors detected** (RX_LEN)

  0 – No length errors were detected.

1 – At least one length error was detected.

- **Frame CRC errors detected** (RX_CRC)

  0 – No frame CRC errors were detected.

  1 – At least one CRC error was detected.

- **Alignment errors detected** (RX_ALIGN)

  0 – No alignment errors were detected.

  1 – At least one alignment error was detected.

- **Frame too long errors detected** (RX_LONG)

  0 – No frame too long errors were detected.

  1 – At least one frame too long error was detected.

- **Frames received OK** (RX_OK)

  This bit can be used to generate an interrupt on the next RX frame.

  0 – No good frames have been received.

  1 – At least one frame has been received OK.

- **Frames received** (RX_COMP)

  0 – No frames were received.

  1 – At least one frame (good or bad) was received.

Ethernet MAC RX Frame Status Interrupt Enable (EMAC_RX_IRQE) Register

The EMAC_RX_IRQE register, shown in Figure 21-36, enables the frame status interrupts.

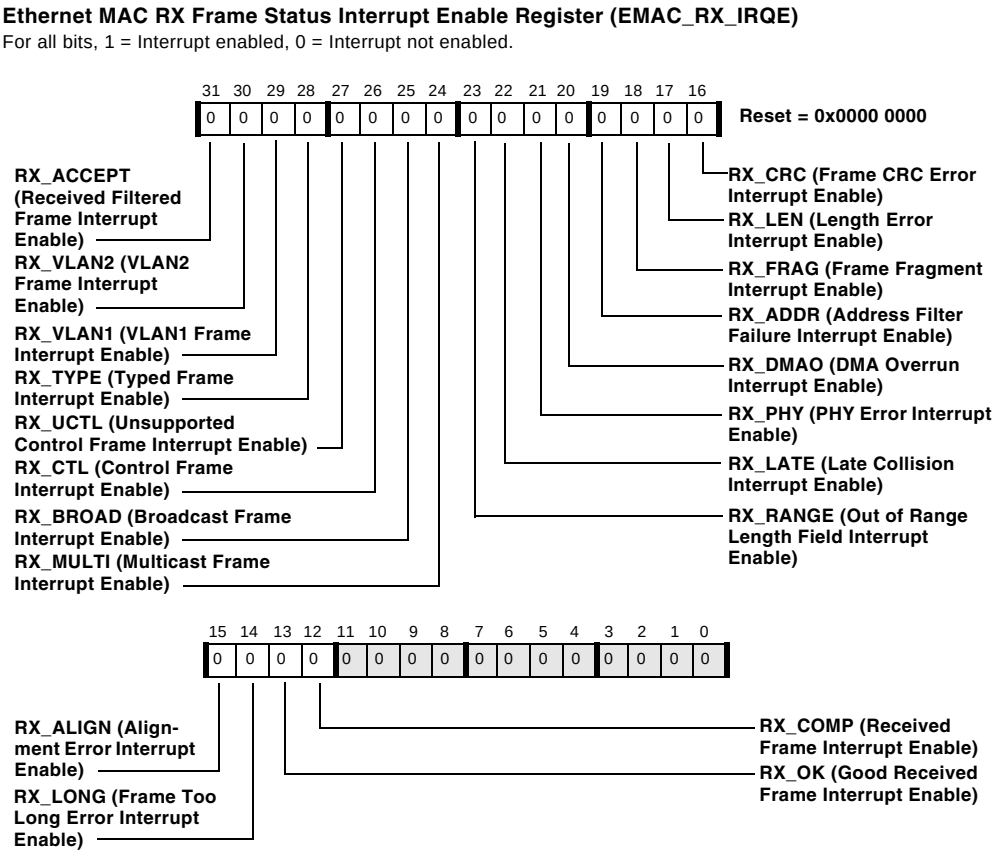

Ethernet MAC RX Frame Status Interrupt Enable Register (EMAC_RX_IRQE)
For all bits, 1 = Interrupt enabled, 0 = Interrupt not enabled.
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
RX_ACCEPT (Received Filtered Frame Interrupt Enable)
RX_VLAN2 (VLAN2 Frame Interrupt Enable)
RX_VLAN1 (VLAN1 Frame Interrupt Enable)
RX_TYPE (Typed Frame Interrupt Enable)
RX_UCTL (Unsupported Control Frame Interrupt Enable)
RX_CTL (Control Frame Interrupt Enable)
RX_BROAD (Broadcast Frame Interrupt Enable)
RX_MULTI (Multicast Frame Interrupt Enable)
RX_CRC (Frame CRC Error Interrupt Enable)
RX_LEN (Length Error Interrupt Enable)
RX_FRAG (Frame Fragment Interrupt Enable)
RX_ADDR (Address Filter Failure Interrupt Enable)
RX_DMAO (DMA Overrun Interrupt Enable)
RX_PHY (PHY Error Interrupt Enable)
RX_LATE (Late Collision Interrupt Enable)
RX_RANGE (Out of Range Length Field Interrupt Enable)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RX_ALIGN (Alignment Error Interrupt Enable)
RX_LONG (Frame Too Long Error Interrupt Enable)
RX_COMP (Received Frame Interrupt Enable)
RX_OK (Good Received Frame Interrupt Enable)

Figure 21-36. EMAC_RX_IRQE Register

Ethernet MAC TX Current Frame Status (EMAC_TX_STAT) Register

The EMAC_TX_STAT register, shown in Figure 21-37, tells the status of the most recently completed transmit frame, including type of error for cases where an error occurred. When the transmit complete bit is set, exactly one of bits 2, 3, 4, 13, or 14 is 1. Bits 1 through 3 indicate the transmit status as defined in IEEE 802.3, section 4.3.2.

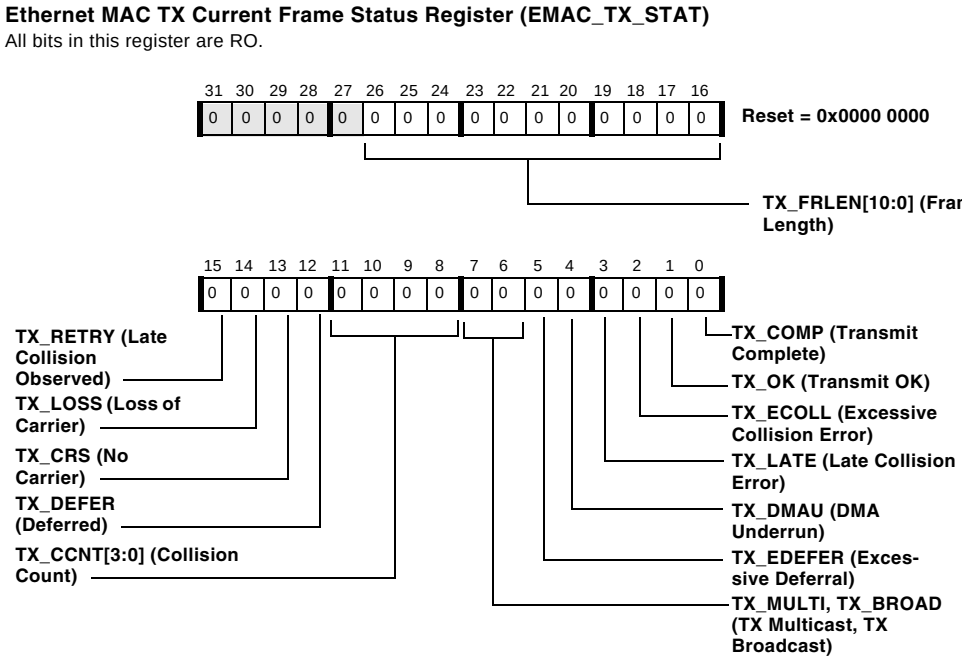

Ethernet MAC TX Current Frame Status Register (EMAC_TX_STAT)
All bits in this register are RO.
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
TX_FRLEN[10:0] (Frar Length)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
TX_RETRY (Late Collision Observed)
TX_LOSS (Loss of Carrier)
TX_CRS (No Carrier)
TX_DEFER (Deferred)
TX_CCNT[3:0] (Collision Count)
TX_COMP (Transmit Complete)
TX_OK (Transmit OK)
TX_ECOLL (Excessive Collision Error)
TX_LATE (Late Collision Error)
TX_DMAU (DMA Underrun)
TX_EDEFER (Excessive Deferral)
TX_MULTI, TX_BROAD (TX Multicast, TX Broadcast)

Figure 21-37. EMAC_TX_STAT Register

Additional information for the EMAC_TX_STAT register bits includes:

- **TX frame length** (TX_FRLEN)

  This field contains the length of the transmit frame in bytes.

- **Late collision observed** (TX_RETRY)

0 – No late collision occurred.

1 – A late collision occurred, but the frame transmission was successful after retry.

- **Loss of carrier** (TX_LOSS)

  0 – No loss of carrier occurred.

  1 – The carrier sense transitioned from asserted to deasserted at some time during the frame transmission. Half-duplex only, MII mode only.

- **No carrier** (TX_CRS)

  0 – CRS was asserted.

  1 – Carrier sense (CRS) was not asserted at any time during frame transmission. Half-duplex only.

- **Deferred** (TX_DEFER)

  0 – Transmission not deferred.

  1 – The transmission was deferred in half-duplex mode because the medium was initially occupied (CRS was asserted) at the time the frame was ready to transmit (after the initial frame data was transferred by DMA to the MAC). Note the deferred status bit should be expected to be 1 on frames that have been retried after early collisions, since the MAC can restart the frame immediately after a collision using data available in its local FIFO. Since the MAC does not need to wait for DMA, the frame data is typically ready for retransmission before TXEN and CRS have deasserted from the prior attempt. Half-duplex only.

- **Collision count** (TX_CCNT)

  This field contains the number of collisions that occurred during frame transmission.

- **TX broadcast, TX multicast** (TX_BROAD, TX_MULTI)

  00 – Unicast address

  01 – Broadcast address

  10 – Group address

  11 – Illegal

- **Excessive deferral** (TX_EDEFER)

  0 – Excessive deferral did not occur.

  1 – The frame transmission was deferred for more than 24,288 bit times or 6072 TX clocks:

  MaxDeferTime = 2 x (MaxUntaggedFrameSize x 8) bits

  If the deferral check (DC) bit in the EMAC_OPMODE register is 1, frame transmission is aborted upon excessive deferral, and both the excessive deferral and excessive collision error status bits are set.

- **DMA underrun** (TX_DMAU)

  0 – No DMA underrun.

  1 – The frame transmission was interrupted by a failure of the FIFO/DMA channel to continuously supply frame data after the start of transmission on the MII/RMII.

- **Late collision error** (TX_LATE)

  0 – No late collision error.

  1 – Frame transmission failed because a collision occurred after the end of the collision window (512 bit times) and the LCRTE bit was clear, disabling frame transmission retry.

- **Excessive collision error** (TX_ECOLL)

0 – No excessive collision error.

1 – Frame transmission failed because too many (16) attempts were interrupted by collisions, or because the frame was deferred for more than the maximum deferral time while the deferral check (DC) control bit was set.

- **Transmit OK** (TX_OK)

  0 – A transmit error occurred.

  1 – There was no transmit error.

- **Transmit complete** (TX_COMP)

  This bit is cleared on reset and when the MAC TX is enabled (TE changes from 0 to 1). In the TX DMA status buffer, this bit is always set to 1 on every status word written via DMA. This bit thus acts as a semaphore, indicating to software that processing of this descriptor pair has been completed.

  0 – The first TX frame is not yet complete.

  1 – The first TX frame is complete.

Ethernet MAC TX Sticky Frame Status (EMAC_TX_STKY) Register

The EMAC_TX_STKY register, shown in Figure 21-38, accumulates state across multiple frames, unless software clears it after every frame.

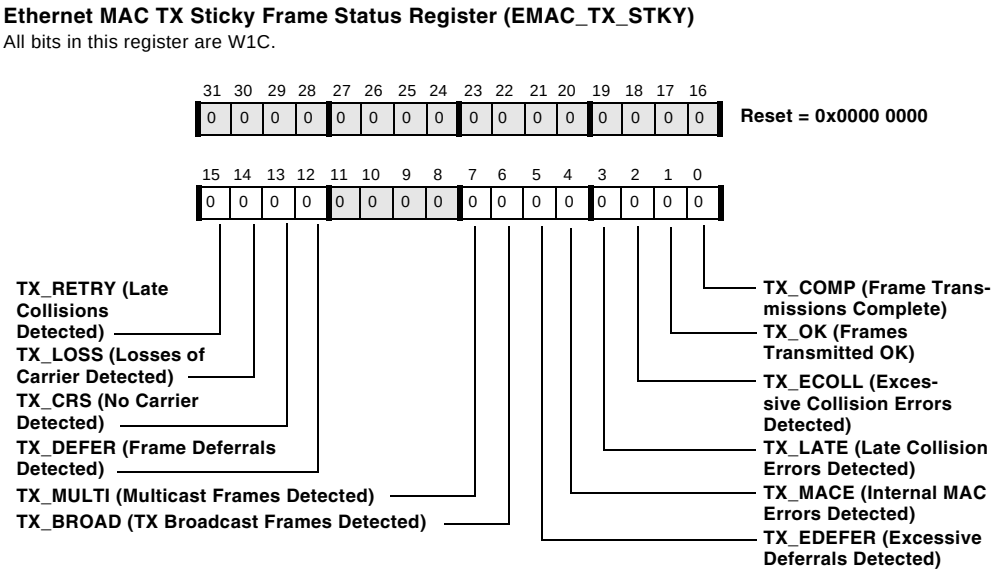

Ethernet MAC TX Sticky Frame Status Register (EMAC_TX_STKY)
All bits in this register are W1C.
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
TX_RETRY (Late Collisions Detected)
TX_LOSS (Losses of Carrier Detected)
TX_CRS (No Carrier Detected)
TX_DEFER (Frame Deferrals Detected)
TX_MULTI (Multicast Frames Detected)
TX_BROAD (TX Broadcast Frames Detected)
TX_COMP (Frame Transmissions Complete)
TX_OK (Frames Transmitted OK)
TX_ECOLL (Excessive Collision Errors Detected)
TX_LATE (Late Collision Errors Detected)
TX_MACE (Internal MAC Errors Detected)
TX_EDEFER (Excessive Deferrals Detected)

Figure 21-38. EMAC_TX_STKY Register

Additional information for the EMAC_TX_STKY register bits includes:

- **Late collisions detected** (TX_RETRY)

  0 – No late collisions were detected.

  1 – At least one late collision was detected on frames successfully transmitted after retry.

- **Losses of carrier detected** (TX_LOSS)

  0 – No losses of carrier were detected.

1 – At least one loss of carrier was detected

- **No carrier detected** (TX_CRS)

  0 – No instances of no carrier were detected.

  1 – At least one occasion of no carrier was detected.

- **Frame deferrals detected** (TX_DEFER)

  0 – No frame deferrals were detected.

  1 – At least one frame deferral was detected.

- **TX multicast frames detected** (TX_MULTI)

  0 – No multicast frames were detected.

  1 – At least one multicast frame was detected.

- **TX broadcast frames detected** (TX_BROAD)

  0 – No broadcast frames were detected.

  1 – At least one broadcast frame was detected.

- **Excessive deferrals detected** (TX_EDEFER)

  0 – No excessive deferrals were detected.

  1 – At least one excessive deferral was detected.

- **Internal MAC errors detected** (TX_MACE)

  0 – No internal MAC errors were detected.

  1 – At least one internal MAC error was detected.

- **Late collision errors detected** (TX_LATE)

  0 – No late collision errors were detected.

1 – At least one late collision error was detected.

- **Excessive collision errors detected** (TX_ECOLL)

  0 – No excessive collision errors were detected.

  1 – At least one excessive collision error detected.

- **Frame transmissions complete** (TX_COMP)

  0 – No frames have been transmitted.

  1 – At least one frame was transmitted.

- **Frames transmitted OK** (TX_OK)

  This bit can be used to generate an interrupt at the completion of each TX frame.

  0 – No good frames have been transmitted.

  1 – At least one frame has been transmitted OK.

Ethernet MAC TX Frame Status Interrupt Enable (EMAC_TX_IRQE) Register

The EMAC_TX_IRQE register, shown in Figure 21-39, is used to enable TX frame status interrupts.

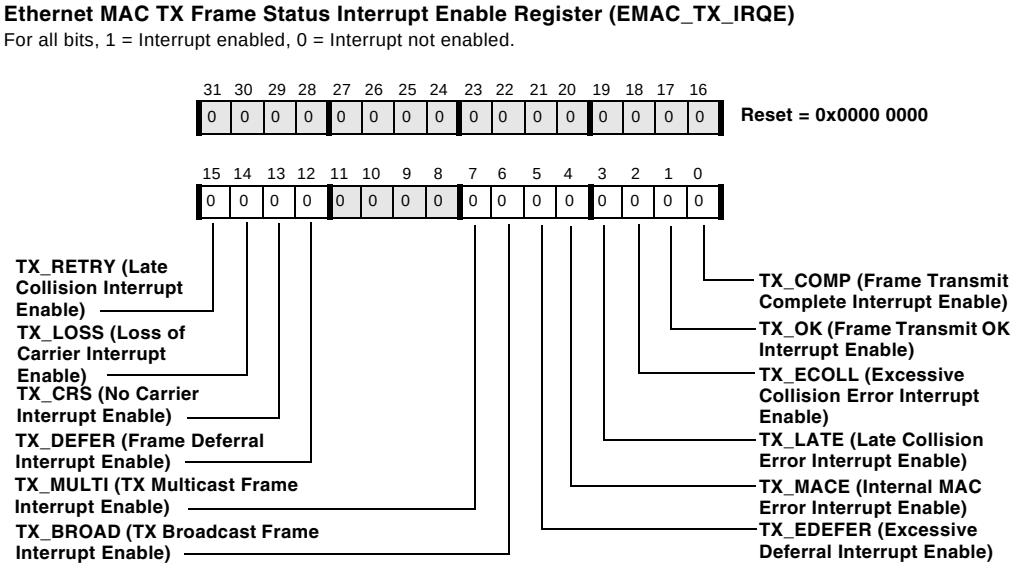

Ethernet MAC TX Frame Status Interrupt Enable Register (EMAC_TX_IRQE)
For all bits, 1 = Interrupt enabled, 0 = Interrupt not enabled.
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
TX_RETRY (Late Collision Interrupt Enable)
TX_LOSS (Loss of Carrier Interrupt Enable)
TX_CRS (No Carrier Interrupt Enable)
TX_DEFER (Frame Deferral Interrupt Enable)
TX_MULTI (TX Multicast Frame Interrupt Enable)
TX_BROAD (TX Broadcast Frame Interrupt Enable)
TX_COMP (Frame Transmit Complete Interrupt Enable)
TX_OK (Frame Transmit OK Interrupt Enable)
TX_ECOLL (Excessive Collision Error Interrupt Enable)
TX_LATE (Late Collision Error Interrupt Enable)
TX_MACE (Internal MAC Error Interrupt Enable)
TX_EDEFER (Excessive Deferral Interrupt Enable)

Figure 21-39. EMAC_TX_IRQE Register

Ethernet MAC MMC RX Interrupt Status (EMAC_MMC_RIRQS) Register

The EMAC_MMC_RIRQS register, shown in Figure 21-40, indicates which of the receive MAC management counters have incremented past one-half of maximum range. Each bit is set from 0 to 1 when the corresponding counter increments from a value less than 0x8000 0000 to a value greater than or equal to 0x8000 0000 (regardless of the state of the EMAC_MMC_RIRQE register). Bits in this register are cleared by writing a 1; writing zero has no effect. For more information, see "MAC Management Counters" on page 21-43.

Ethernet MAC MMC RX Interrupt Status Register (EMAC_MMC_RIRQS)

All bits are W1C. For all bits, 1 = Interrupt occurred, 0 = Interrupt did not occur.

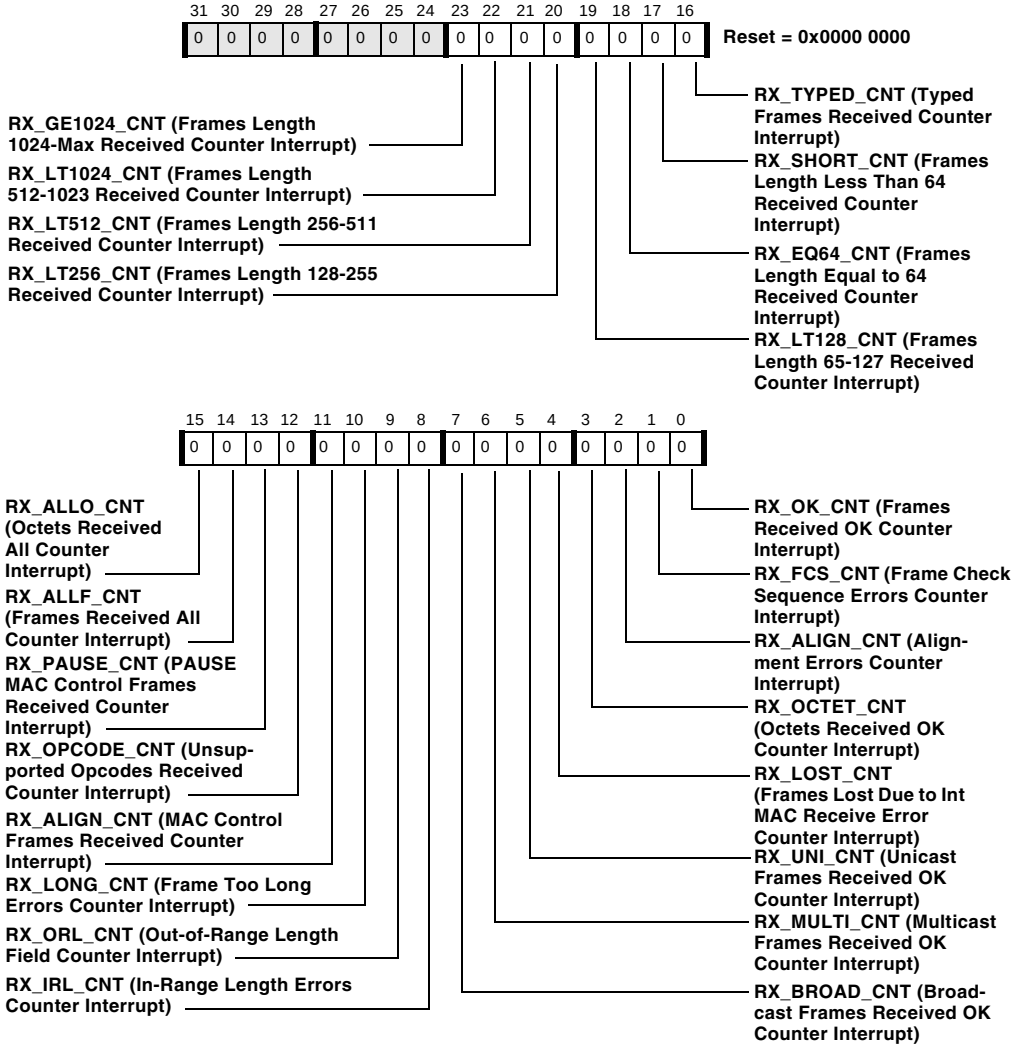

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
RX_TYPED_CNT (Typed Frames Received Counter Interrupt)
RX_SHORT_CNT (Frames Length Less Than 64 Received Counter Interrupt)
RX_EQ64_CNT (Frames Length Equal to 64 Received Counter Interrupt)
RX_LT128_CNT (Frames Length 65-127 Received Counter Interrupt)
RX_GE1024_CNT (Frames Length 1024-Max Received Counter Interrupt)
RX_LT1024_CNT (Frames Length 512-1023 Received Counter Interrupt)
RX_LT512_CNT (Frames Length 256-511 Received Counter Interrupt)
RX_LT256_CNT (Frames Length 128-255 Received Counter Interrupt)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RX_OK_CNT (Frames Received OK Counter Interrupt)
RX_FCS_CNT (Frame Check Sequence Errors Counter Interrupt)
RX_ALIGN_CNT (Alignment Errors Counter Interrupt)
RX_OCTET_CNT (Octets Received OK Counter Interrupt)
RX_LOST_CNT (Frames Lost Due to Int MAC Receive Error Counter Interrupt)
RX_UNI_CNT (Unicast Frames Received OK Counter Interrupt)
RX_MULTI_CNT (Multicast Frames Received OK Counter Interrupt)
RX_BROAD_CNT (Broadcast Frames Received OK Counter Interrupt)
RX_ALLO_CNT (Octets Received All Counter Interrupt)
RX_ALLF_CNT (Frames Received All Counter Interrupt)
RX_PAUSE_CNT (PAUSE MAC Control Frames Received Counter Interrupt)
RX_OPCODE_CNT (Unsupported Opcodes Received Counter Interrupt)
RX_ALIGN_CNT (MAC Control Frames Received Counter Interrupt)
RX_LONG_CNT (Frame Too Long Errors Counter Interrupt)
RX_ORL_CNT (Out-of-Range Length Field Counter Interrupt)
RX_IRL_CNT (In-Range Length Errors Counter Interrupt)

Figure 21-40. EMAC_MMC_RIRQS Register

Ethernet MAC MMC RX Interrupt Enable (EMAC_MMC_RIRQE) Register

The EMAC_MMC_RIRQE register, shown in Figure 21-41, indicates which of the receive MAC management counters are enabled to signal an MMCINT interrupt when they increment past one-half of maximum range.

If a given counter's interrupt is not enabled, and that counter passes 0x8000 0000, then the counter's interrupt status bit is set to 1 but this does not cause the MMCINT interrupt to be signalled. If the corresponding interrupt enable bit is later written to 1, the MMCINT Ethernet event interrupt is signalled immediately.

Ethernet MAC MMC RX Interrupt Enable Register (EMAC_MMC_RIRQE)

For all bits, 1 = Interrupt enabled, 0 = Interrupt not enabled.

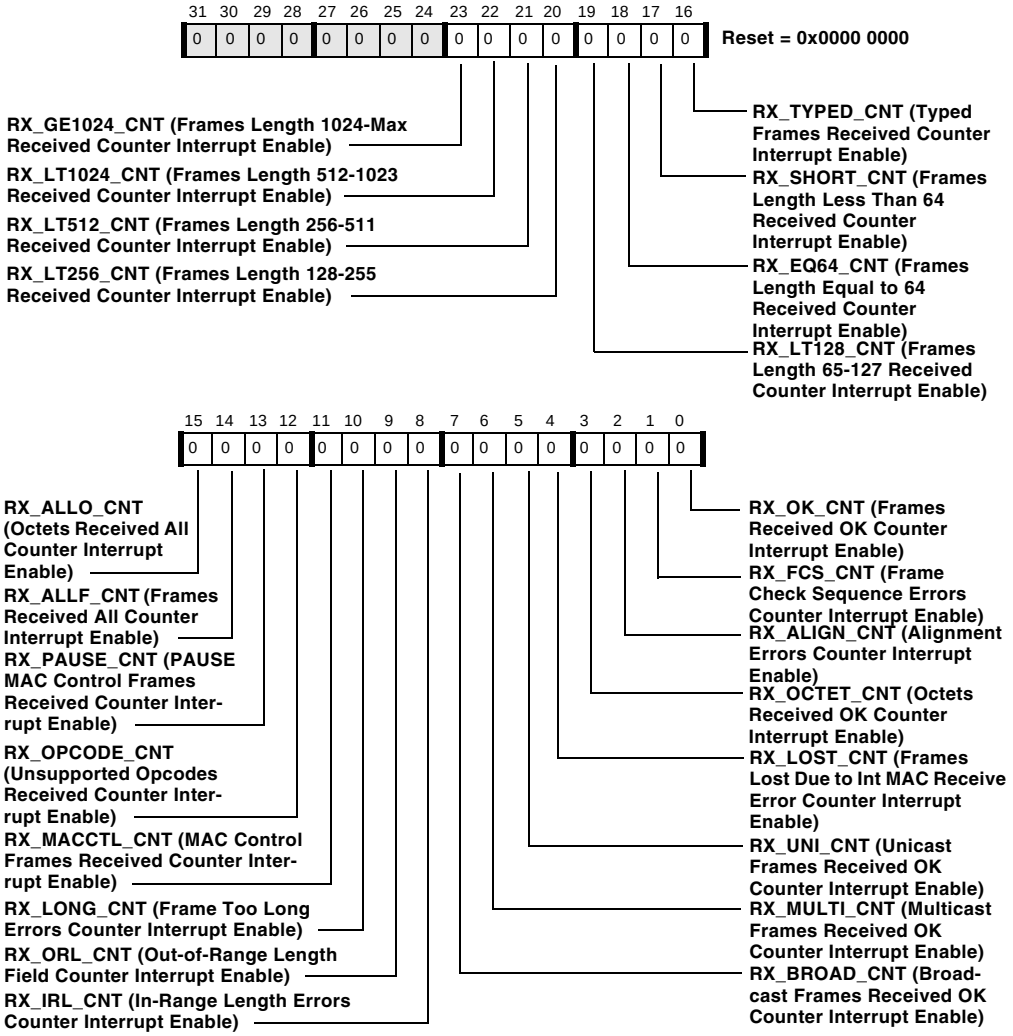

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
RX_GE1024_CNT (Frames Length 1024-Max Received Counter Interrupt Enable)
RX_LT1024_CNT (Frames Length 512-1023 Received Counter Interrupt Enable)
RX_LT512_CNT (Frames Length 256-511 Received Counter Interrupt Enable)
RX_LT256_CNT (Frames Length 128-255 Received Counter Interrupt Enable)
RX_TYPED_CNT (Typed Frames Received Counter Interrupt Enable)
RX_SHORT_CNT (Frames Length Less Than 64 Received Counter Interrupt Enable)
RX_EQ64_CNT (Frames Length Equal to 64 Received Counter Interrupt Enable)
RX_LT128_CNT (Frames Length 65-127 Received Counter Interrupt Enable)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RX_ALLO_CNT (Octets Received All Counter Interrupt Enable)
RX_ALLF_CNT (Frames Received All Counter Interrupt Enable)
RX_PAUSE_CNT (PAUSE MAC Control Frames Received Counter Interrupt Enable)
RX_OPCODE_CNT (Unsupported Opcodes Received Counter Interrupt Enable)
RX_MACCTL_CNT (MAC Control Frames Received Counter Interrupt Enable)
RX_LONG_CNT (Frame Too Long Errors Counter Interrupt Enable)
RX_ORL_CNT (Out-of-Range Length Field Counter Interrupt Enable)
RX_IRL_CNT (In-Range Length Errors Counter Interrupt Enable)
RX_OK_CNT (Frames Received OK Counter Interrupt Enable)
RX_FCS_CNT (Frame Check Sequence Errors Counter Interrupt Enable)
RX_ALIGN_CNT (Alignment Errors Counter Interrupt Enable)
RX_OCTET_CNT (Octets Received OK Counter Interrupt Enable)
RX_LOST_CNT (Frames Lost Due to Int MAC Receive Error Counter Interrupt Enable)
RX_UNI_CNT (Unicast Frames Received OK Counter Interrupt Enable)
RX_MULTI_CNT (Multicast Frames Received OK Counter Interrupt Enable)
RX_BROAD_CNT (Broadcast Frames Received OK Counter Interrupt Enable)

Figure 21-41. EMAC_MMC_RIRQE Register

Ethernet MAC MMC TX Interrupt Status (EMAC_MMC_TIRQS) Register

The EMAC_MMC_TIRQS register, shown in Figure 21-42, indicates which of the transmit MAC management counters have incremented past one-half of maximum range. Each bit is set from 0 to 1 when the corresponding counter increments from a value less than 0x8000 0000 to a value greater than or equal to 0x8000 0000 (regardless of the state of the EMAC_MMC_TIRQE register). Bits in this register are cleared by writing a 1; writing zero has no effect. For more information, see "MAC Management Counters" on page 21-43.

Ethernet MAC MMC TX Interrupt Status Register (EMAC_MMC_TIRQS)

All bits are W1C. For all bits, 1 = Interrupt occurred, 0 = Interrupt did not occur.

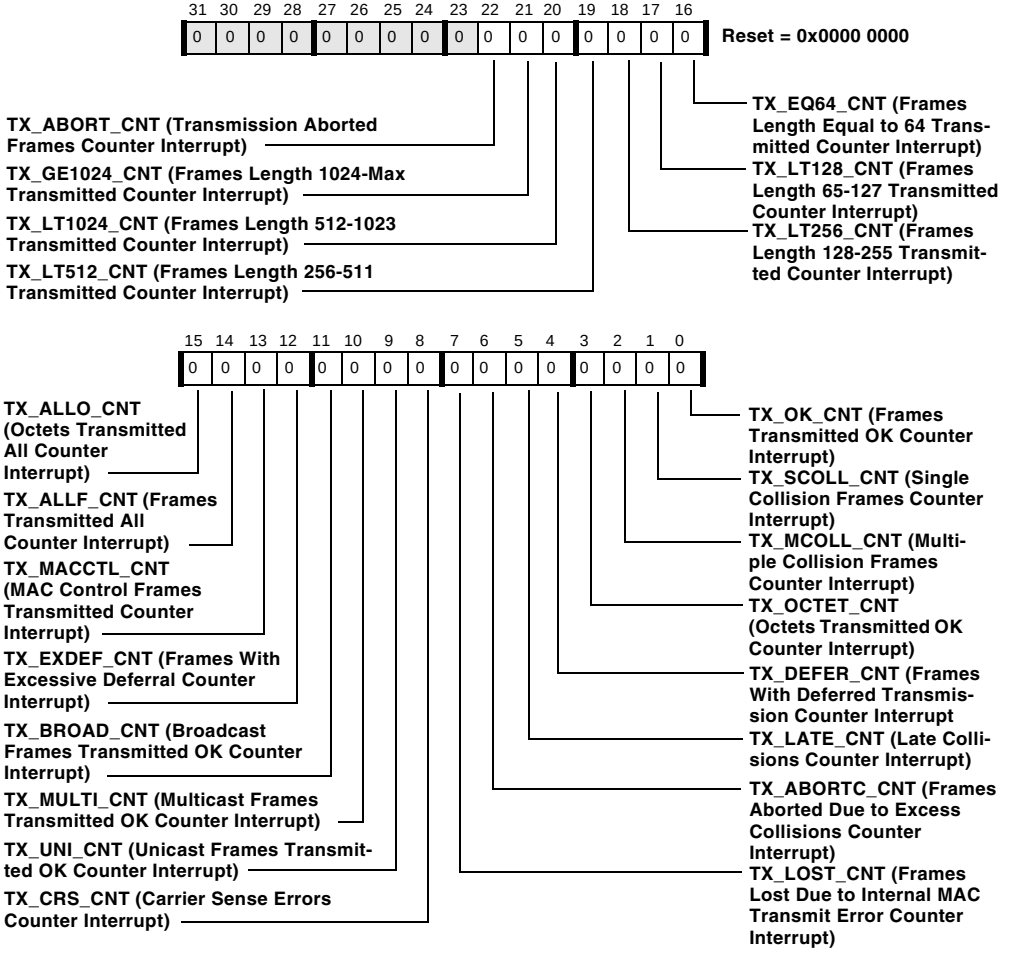

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
TX_ABORT_CNT (Transmission Aborted Frames Counter Interrupt)
TX_GE1024_CNT (Frames Length 1024-Max Transmitted Counter Interrupt)
TX_LT1024_CNT (Frames Length 512-1023 Transmitted Counter Interrupt)
TX_LT512_CNT (Frames Length 256-511 Transmitted Counter Interrupt)
TX_EQ64_CNT (Frames Length Equal to 64 Transmitted Counter Interrupt)
TX_LT128_CNT (Frames Length 65-127 Transmitted Counter Interrupt)
TX_LT256_CNT (Frames Length 128-255 Transmitted Counter Interrupt)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
TX_ALLO_CNT (Octets Transmitted All Counter Interrupt)
TX_ALLF_CNT (Frames Transmitted All Counter Interrupt)
TX_MACCTL_CNT (MAC Control Frames Transmitted Counter Interrupt)
TX_EXDEF_CNT (Frames With Excessive Deferral Counter Interrupt)
TX_BROAD_CNT (Broadcast Frames Transmitted OK Counter Interrupt)
TX_MULTI_CNT (Multicast Frames Transmitted OK Counter Interrupt)
TX_UNI_CNT (Unicast Frames Transmitted OK Counter Interrupt)
TX_CRS_CNT (Carrier Sense Errors Counter Interrupt)
TX_OK_CNT (Frames Transmitted OK Counter Interrupt)
TX_SCOLL_CNT (Single Collision Frames Counter Interrupt)
TX_MCOLL_CNT (Multiple Collision Frames Counter Interrupt)
TX_OCTET_CNT (Octets Transmitted OK Counter Interrupt)
TX_DEFER_CNT (Frames With Deferred Transmission Counter Interrupt
TX_LATE_CNT (Late Collisions Counter Interrupt)
TX_ABORTC_CNT (Frames Aborted Due to Excess Collisions Counter Interrupt)
TX_LOST_CNT (Frames Lost Due to Internal MAC Transmit Error Counter Interrupt)

Figure 21-42. EMAC_MMC_TIRQS Register

Ethernet MAC MMC TX Interrupt Enable (EMAC_MMC_TIRQE) Register

The EMAC_MMC_TIRQE register, shown in Figure 21-43, indicates which of the transmit MAC management counters are enabled to signal an MMCINT interrupt when they increment past one-half of maximum range.

If a given counter's interrupt is not enabled, and that counter passes 0x8000 0000, then the counter's interrupt status bit is set to 1 but this does not cause the MMCINT interrupt to be signalled. If the corresponding interrupt enable bit is later written to 1, the MMCINT Ethernet event interrupt is signalled immediately.

Ethernet MAC MMC TX Interrupt Enable Register (EMAC_MMC_TIRQE)

For all bits, 1 = Interrupt enabled, 0 = Interrupt not enabled.

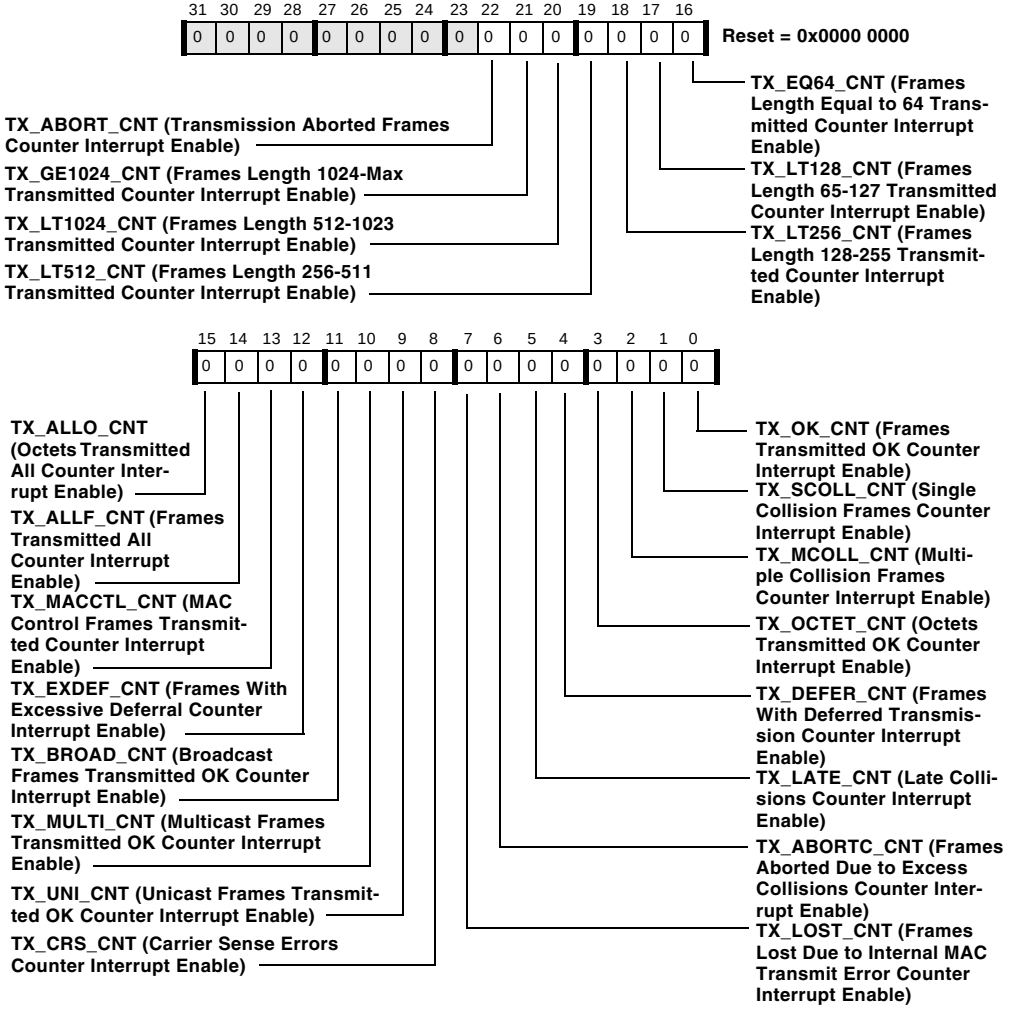

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
TX_EQ64_CNT (Frames Length Equal to 64 Transmitted Counter Interrupt Enable)
TX_LT128_CNT (Frames Length 65-127 Transmitted Counter Interrupt Enable)
TX_LT256_CNT (Frames Length 128-255 Transmitted Counter Interrupt Enable)
TX_ABORT_CNT (Transmission Aborted Frames Counter Interrupt Enable)
TX_GE1024_CNT (Frames Length 1024-Max Transmitted Counter Interrupt Enable)
TX_LT1024_CNT (Frames Length 512-1023 Transmitted Counter Interrupt Enable)
TX_LT512_CNT (Frames Length 256-511 Transmitted Counter Interrupt Enable)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
TX_OK_CNT (Frames Transmitted OK Counter Interrupt Enable)
TX_SCOLL_CNT (Single Collision Frames Counter Interrupt Enable)
TX_MCOLL_CNT (Multiple Collision Frames Counter Interrupt Enable)
TX_OCTET_CNT (Octets Transmitted OK Counter Interrupt Enable)
TX_DEFER_CNT (Frames With Deferred Transmission Counter Interrupt Enable)
TX_LATE_CNT (Late Collisions Counter Interrupt Enable)
TX_ABORTC_CNT (Frames Aborted Due to Excess Collisions Counter Interrupt Enable)
TX_LOST_CNT (Frames Lost Due to Internal MAC Transmit Error Counter Interrupt Enable)
TX_ALLO_CNT (Octets Transmitted All Counter Interrupt Enable)
TX_ALLF_CNT (Frames Transmitted All Counter Interrupt Enable)
TX_MACCTL_CNT (MAC Control Frames Transmitted Counter Interrupt Enable)
TX_EXDEF_CNT (Frames With Excessive Deferral Counter Interrupt Enable)
TX_BROAD_CNT (Broadcast Frames Transmitted OK Counter Interrupt Enable)
TX_MULTI_CNT (Multicast Frames Transmitted OK Counter Interrupt Enable)
TX_UNI_CNT (Unicast Frames Transmitted OK Counter Interrupt Enable)
TX_CRS_CNT (Carrier Sense Errors Counter Interrupt Enable)

Figure 21-43. EMAC_MMC_TIRQE Register

MAC Management Counter Registers

The MAC Management Counter (MMC) block register group consists of a number of 32-bit unsigned counter registers that gather statistical data regarding the operation of the MAC. The MAC management counter registers update automatically at the completion of frame transmit and receive, whenever the MMCE bit in the EMAC_MMC_CTL register is set. Counters contain a 32-bit unsigned value, and may be configured to saturate at 0xFFFF FFFF (CROLL = 0) or to wrap around to zero (CROLL = 1). Counters cannot be written directly, but can be collectively reset to zero by writing 1 to the RSTC bit, or they can be programmed for clear-on-read behavior by setting CCOR to 1. The reset value for all MMC registers is 0x0000 0000. See Table 21-10 on page 21-54 for more information.

Each of these counters can be set up to generate interrupts when they reach half of the maximum unsigned 32-bit value. This functionality is described in detail in the section entitled "Ethernet Event Interrupts" on page 21-39.

MAC Management Counters Control (EMAC_MMC_CTL) Register

The EMAC_MMC_CTL register, shown in Figure 21-44, is used to globally configure all MMC counter registers.

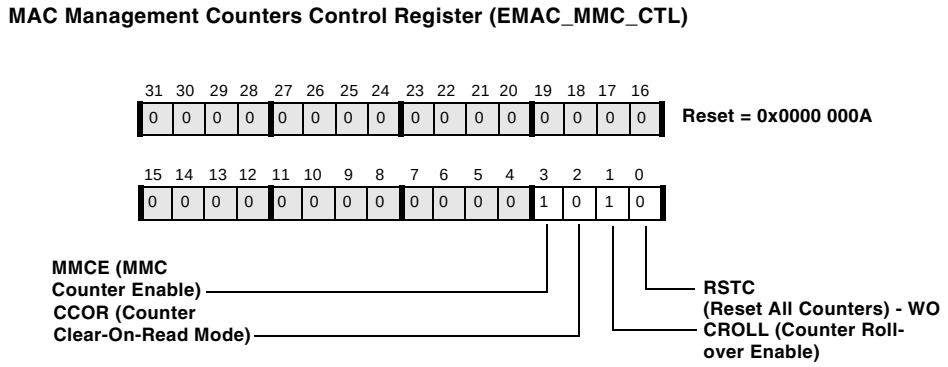

MAC Management Counters Control Register (EMAC_MMC_CTL)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 000A
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
MMCE (MMC Counter Enable)
CCOR (Counter Clear-On-Read Mode)
RSTC (Reset All Counters) - WO
CROLL (Counter Roll-over Enable)

Figure 21-44. EMAC_MMC_CTL Register

Additional information for the EMAC_MMC_CTL register bits includes:

- **MMC counter enable** (MMCE)

  Setting this bit turns on all the MMC counters, which update on every frame transmission or reception.

  0 – MMC counters are not enabled. Counters retain their values but are not updated.

  1 – MMC counters are enabled.

- **Counter clear-on-read mode** (CCOR)

  0 – Counters are not in clear-on-read mode. Reads do not affect counter contents.

  1 – Counters are in clear-on-read mode. The contents of each counter is reset each time it is read by the application.

- **Counter rollover enable** (CROLL)

  0 – Counter rollover is not enabled. All MMC registers saturate upon reaching 0xFFFF FFFF.

  1 – Counter rollover is enabled. This causes all MMC counters to wrap around to zero when the count exceeds the maximum 32-bit value of 0xFFFF FFFF.

- **Reset all counters** (RSTC)

  Writing a 1 to this bit at any time globally resets all MMC counters.

  0 – Do not reset all counters.

  1 – Globally clear all MMC counters.

Programming Examples

This section gives a general overview of the functionality of an Ethernet MAC driver. All necessary steps for reproducing and understanding this interface are explained with code listings and accompanying text. These code listings are similar to the driver model supported by VisualDSP++ and are mainly written in C. Data transfers over the MAC with DMA are explained in Figure 21-5 on page 21-11 and Figure 21-7 on page 21-23, which show receive and transmit DMA operations. Please examine these figures carefully—the code listings reproduce this kind of "linked list" in the form of C structures. Also provided are code listings that describe accessing an external PHY via the station management (MIM) block. All macros that are not explained in this section can be found in the `cdefBF5xx.h` and `defBF5xx.h` header files of VisualDSP++.

The code examples in this section (Listing 21-1 through Listing 21-9) show basic functions and structures. The management counter registers and the interrupt settings are advanced functions and are not covered

here. There are many counter registers which are accessible by polling of the appropriate register or using interrupt service routines. The EMAC_SYSCTL and EMAC_SYSTAT register should be used to configure the Ethernet MAC interrupt capabilities. See Figure 21-12 on page 21-39 for a detailed description of the MAC interrupts.

Ethernet Structures

Listing 21-1. Type Definition

```
// type definitions
typedef unsigned long  int   u32;
typedef unsigned short int   u16;
typedef unsigned       char  u8;
typedef volatile u32         reg32;
typedef volatile u16         reg16;
```

The type definitions are placed here to help with reading of the following code.

Listing 21-2. DMA Configuration

```
typedef struct ADI_DMA_CONFIG_REG {
  u16 b_DMAEN:1;     /* 0      Enabled */
  u16 b_WNR:1;        /* 1     Direction */
  u16 b_WDSIZE:2;     /* 2:3   Transfer word size */
  u16 b_DMA2D:1;      /* 4     DMA mode */
  u16 b_SYNC:1;    /* 5     Retain FIFO */
  u16 b_DI_SEL:1;     /* 6     Data interrupt timing select */
  u16 b_DI_EN:1;      /* 7     Data interrupt enabled */
  u16 b_NDSIZE:4;     /* 8:11  Flex descriptor size */
  u16 b_FLOW:3;       /* 12:14 Flow */
} ADI_DMA_CONFIG_REG;
```

A convenient way to handle the DMA properties in a “linked list” is to use structures, because each set should be assigned to the appropriate DMA descriptor. Listing 21-3 shows a structure used to manage DMA descriptors. Before jumping to the next descriptor, like 1A-1B-2A-2B-1C in Figure 21-5 on page 21-11 and Figure 21-7 on page 21-23, the structure `ADI_DMA_CONFIG_REG` immediately loads to the DMA register before starting its DMA transfer.

Listing 21-3. DMA Descriptor

```
typedef struct dma_descriptor {
  struct dma_descriptor*  NEXT_DESC_PTR;
  u32                     START_ADDR;
  ADI_DMA_CONFIG_REG      CONFIG;
} DMA_DESCRIPTOR;
```

The structure shown in Listing 21-3 shows how it is possible to create a “linked list” of DMAs. The `START_ADDR` points to the data and the `ADI_DMA_CONFIG_REG` structure (shown in Listing 21-2) holds all the necessary settings.

Structures like these are convenient for handling Ethernet streams, because they allow the programmer to simply call members of the structure instead of extracting meaningful items through array offsets. This structure, shown in Listing 21-4, is mirrored in the Ethernet MAC header with additional NoBytes.

Listing 21-4. Ethernet Frame Buffer

```
typedef struct adi_ether_frame_buffer {
  u16   NoBytes;  /* the no. of following bytes */
  u8    Dest[6];  /* destination MAC address */
  u8    Srce[6];  /* source MAC address */
  u16   LTfield;  /* length/type field */
```

```
  u8    Data[0];  /* payload bytes */
} ADI_ETHER_FRAME_BUFFER;
```

The ADI_ETHER_BUFFER structure in Listing 21-5, Top Level Structure, covers all the above structures and shows the general framework as described in Figure 21-5 on page 21-11 and Figure 21-7 on page 21-23. The two Dma[2] structures are needed for descriptors 1A,1B and 2A,2B. The pointer *frmData represents the payload of the frame, which has a specific number of bytes (as dictated by the NoBytes structure member). This is relevant only in transmit mode—in receive mode the driver will not touch this NoBytes variable. To ease programming by keeping the transmit and receive structures the same, the MAC can pad the first 16-bit word (that is, the data corresponding to the NoBytes structure member) with zeros if the RXDWA bit in EMAC_SYSCTL is 1. The *pNext and *pPrev pointers are necessary for creating a "linked list." The IPHdrChksum and IPPayloadChksum are available in case the Ethernet MAC is set to calculate this. See the RXCKS bit in the EMAC_SYSCTL register (shown in Figure 21-32 on page 21-90). These two variables are relevant only in receive mode of the Ethernet MAC. The StatusWord variable holds the EMAC_RX_STAT register value in receive mode and holds the EMAC_TX_STAT register value in transmit mode.

Listing 21-5. Top Level Structure

```
typedef struct adi_ether_buffer {
  DMA_DESCRIPTOR  Dma[2];  /* first for the frame, second for the
status */
  ADI_ETHER_FRAME_BUFFER  *FrmData;  /* pointer to data */
  struct adi_ether_buffer  *pNext;  /* next buffer */
  struct adi_ether_buffer  *pPrev;  /* prev buffer */
  u16  IPHdrChksum;      /* the IP header checksum */
  u16  IPPayloadChksum;  /* the IP header and payload checksum */
  u32  StatusWord;       /* the frame status word */
} ADI_ETHER_BUFFER;
```

MAC Address Setup

Write EMAC_ADDRLO and EMAC_ADDRHI in the initialization routine of the Ethernet MAC, as shown in Listing 21-6. The Ethernet MAC address is a unique number and may not be used twice. See the IEEE Std. 802.3-2002 specification for further information.

Listing 21-6. MAC Address Setup

```
// MAC address
u8 SrcAddr[6] = {0x5A,0xD4,0x9A,0x48,0xDE,0xAC};

// function
void SetupMacAddr(u8 *MACaddr)
{
  *pEMAC_ADDRLO = *(u32 *)&MACaddr[0];
  *pEMAC_ADDRHI = *(u16 *)&MACaddr[4];
}
// function call
SetupMacAddr(SrcAddr);
```

PHY Control Routines

The EMAC_STAAD register provides the option of either polling the STABUSY bit or getting an interrupt during each MIM block access. The function in Listing 21-7 polls the STABUSY bit and should be placed after each read or write command to the PHY register.

Listing 21-7. Poll MIM Block

```
//
/* Wait until the previous MDC/MDIO transaction has completed */
//
void PollMdcDone(void)
```

```
{
  /* poll the STABUSY bit */
  while(*pEMAC_STAADD & STABUSY)
}
```

Shown in Listing 21-8, the SET_PHYAD and SET_REGAD macros shift the PHYAddr and RegAddr values to the appropriate field within the EMAC_STAADD register. The other macros STAOP, STAIE, and STABUSY, also set bits in the EMAC_STAADD register. Use of the STAOP macro controls the read and write transfer of the MIM block.

Listing 21-8. Write Access to the PHY

```
//
/*Write an off-chip register in a PHY through the MDC/MDIO port*/
//
void WrPHYReg(u16 PHYAddr, u16 RegAddr, u16 Data)
{
  PollMdcDone();
  *pEMAC_STADAT = Data;
  *pEMAC_STAADD = SET_PHYAD(PHYAddr) |\
                  SET_REGAD(RegAddr) |\
                  STAOP | STABUSY;
}
```

The data in the EMAC_STADAT register is immediately shifted out after a write to the EMAC_STAADD register. See Figure 21-4 on page 21-9.

The function in Listing 21-9 shows how PHY data is read over the MIM function block of the MAC. First, the STABUSY bit of the EMAC_STAADD register will be polled until no other function is using the MIM block. The PHY address and register address is sent over the MIM block. Then, the STABUSY bit is polled again, before the data is finally read through the EMAC_STADAT register.

Listing 21-9. Read Access to the PHY

```
//
/*Read an off-chip register in a PHY through the MDC/MDIO port*/
//
  u16  RdPHYReg(u16 PHYAddr, u16 RegAddr)
{
  u16  Data;
  PollMdcDone();

  *pEMAC_STAADD =  SET_PHYAD(PHYAddr) |\
                   SET_REGAD(RegAddr) |\
                   STABUSY;

  PollMdcDone();
  Data = (u16)*pEMAC_STADAT;

  return Data;
}
```

A complete PHY initialization also requires the initialization of the station management clock, which is described in detail in the section "MII Station Management" on page 21-48. The three PHY functions included in this section (write, read, and poll) and the initialization routine of the station management clock are the minimum requirements for setup and control of any PHYs.

Unique Behavior for the ADSP-BF52x Processor

None.

22 SPI-COMPATIBLE PORT CONTROLLER

This chapter describes the serial peripheral interface (SPI) port. Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF52x

For details regarding the number of SPIs for the ADSP-BF52x product, please refer to the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet*.

For SPI DMA channel assignments, refer to Table 6-7 on page 6-106 in Chapter 6, “Direct Memory Access”.

For SPI interrupt vector assignments, refer to Table 5-3 on page 5-18 in Chapter 5, “System Interrupts”.

To determine how each of the SPIs is multiplexed with other functional pins, refer to Table 9-2 on page 9-5 through Table 9-5 on page 9-9 in Chapter 9, “General-Purpose Ports”.

For a list of MMR addresses for each SPI, refer to Appendix A, “System MMR Assignments”.

SPI behavior for the ADSP-BF52x that differs from the general information in this chapter can be found in the section “Unique Behavior for the ADSP-BF52x Processor” on page 22-53.

Overview

The SPI port provides an I/O interface to a wide variety of SPI-compatible peripheral devices.

With a range of configurable options, the SPI port provides a glueless hardware interface with other SPI-compatible devices. SPI is a four-wire interface consisting of two data signals, a device select signal, and a clock signal. SPI is a full-duplex synchronous serial interface, supporting master modes, slave modes, and multimaster environments. The SPI-compatible peripheral implementation also supports programmable bit rate and clock phase/polarities. The SPI features the use of open drain drivers to support the multimaster scenario and to avoid data contention.

Features

The SPI includes these features:

- Full duplex, synchronous serial interface
- Supports 8- or 16-bit word sizes
- Programmable baud rate, clock phase, and polarity
- Supports multimaster environments
- Integrated DMA controller
- Double-buffered transmitter and receiver
- One SPI device select input and multiple chip select outputs
- Programmable shift direction of MSB or LSB first
- Interrupt generation on mode fault, overflow, and underflow
- Shadow register to aid debugging

Typical SPI-compatible peripheral devices that can be used to interface to the SPI-compatible interface include:

- Other CPUs or microcontrollers
- Codecs
- A/D converters
- D/A converters
- Sample rate converters
- SP/DIF or AES/EBU digital audio transmitters and receivers
- LCD displays
- Shift registers
- FPGAs with SPI emulation

Interface Overview

Figure 22-1 on page 22-4 provides a block diagram of the SPI. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the SCK rate, to and from other SPI devices. SPI data is transmitted and received at the same time through the use of a shift register. When an SPI transfer occurs, data is simultaneously transmitted (shifted serially out of the shift register) as new data is received (shifted

serially into the other end of the same shift register). The SCK synchronizes the shifting and sampling of the data on the two serial data pins.

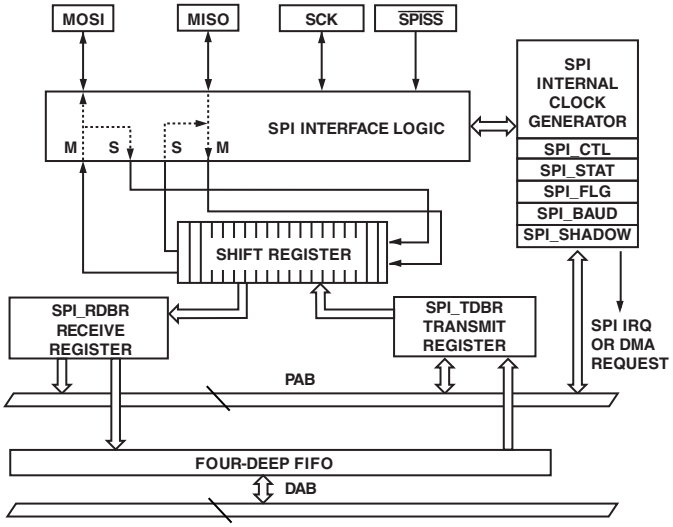


Figure 22-1. SPI Block Diagram

External Interface

SPI Clock Signal (SCK)

The SCK signal is the serial clock signal. This control signal is driven by the master and controls the rate at which data is transferred. The master may transmit data at a variety of bit rates. The SCK signal cycles once for each bit transmitted. It is an output signal if the device is configured as a master, and an input signal if the device is configured as a slave.

The SCK is a gated clock that is active during data transfers only for the length of the transferred word. The number of active clock edges is equal to the number of bits driven on the data lines. Slave devices ignore the serial clock if the SPISS input is driven inactive (high).

The SCK is used to shift out and shift in the data driven on the MISO and MOSI lines. Clock polarity and clock phase relative to data are programmable in the SPI_CTL register and define the transfer format.

Master-Out, Slave-In (MOSI) Signal

The master-out, slave-in (MOSI) signal is one of the bidirectional I/O data pins. If the processor is configured as a master, the MOSI pin transmits data out. If the processor is configured as a slave, the MOSI pin receives data in. In an SPI interconnection, the data is shifted out from the MOSI output pin of the master and shifted into the MOSI input(s) of the slave(s).

Master-In, Slave-Out (MISO) Signal

The master-in, slave-out (MISO) signal is one of the bidirectional I/O data pins. If the processor is configured as a master, the MISO pin receives data in. If the processor is configured as a slave, the MISO pin transmits data out. In an SPI interconnection, the data is shifted out from the MISO output pin of the slave and shifted into the MISO input pin of the master.

Only one slave is allowed to transmit data at any given time.

The SPI configuration example in Figure 22-2 illustrates how the processor can be used as the slave SPI device. The 8-bit host microcontroller is the SPI master.

The processor can be booted through its SPI interface to allow user application code and data to be downloaded before runtime.

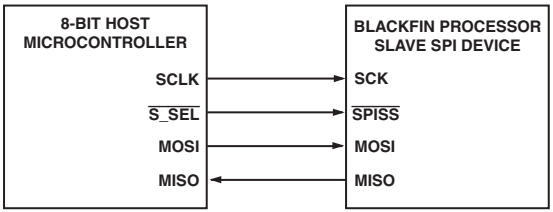


Figure 22-2. Blackfin Processor as Slave SPI Device

SPI Slave Select Input Signal (SPISS)

The SPISS signal is the SPI slave select input signal. This is an active-low signal used to enable a processor when it is configured as a slave device. This input-only pin behaves like a chip select and is provided by the master device for the slave devices. For a master device, it can act as an error signal input in a multimaster environment. In multimaster mode, if the SPISS input signal of a master is asserted (driven low), and the PSSE bit in the SPI_CTL register is enabled, an error has occurred. This means that another device is also trying to be the master device.

The enable lead time (T1), the enable lag time (T2), and the sequential transfer delay time (T3) each must always be greater than or equal to one-half the SCK period. See Figure 22-3 on page 22-7. The minimum time between successive word transfers (T4) is two SCK periods. This is measured from the last active edge of SCK of one word to the first active edge of SCK of the next word. This is independent of the configuration of the SPI (CPHA, MSTR, and so on).

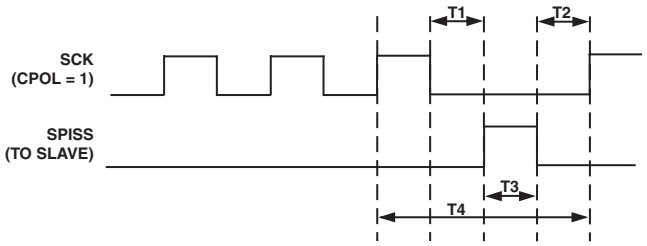


Figure 22-3. SPI Timing

For a master device with CPHA = 0, the slave select output is inactive (high) for at least one-half the SCK period. In this case, T1 and T2 will each always be equal to one-half the SCK period.

SPI Slave Select Enable Output Signals

When operating in master mode, Blackfin processors may use any GPIO pin to enable individual SPI slave devices by software. In addition, the SPI module provides hardware support to generate up to seven slave select enable signals automatically (depending upon the configuration of the specific processor). See Figure 22-14 on page 22-38 for details.

These signals are always active low in the SPI protocol. Since the respective pins are not driven during reset, it is recommended to pull them up by a resistor.

If enabled as a master, the SPI uses the SPI_FLG register to enable general-purpose port pins to be used as individual slave select lines. Before manipulating this register, the port pins that are to be used as SPI slave-select outputs must first be configured as such. To work as SPI output pins, the port pins must be enabled for use by SPI in the appropriate PORT_MUX register.

In slave mode, the SPI_FLG bits have no effect, and each SPI uses the SPISS input as a slave select. Just as in the master mode case, the port pin

associated with SPISS must first be configured appropriately before use. Figure 22-14 on page 22-38 shows the SPI_FLG register diagram.

Slave Select Inputs

If the SPI is in slave mode, SPISS acts as the slave select input. When enabled as a master, SPISS can serve as an error detection input for the SPI in a multimaster environment. The PSSE bit in SPI_CTL enables this feature. When PSSE = 1, the SPISS input is the master mode error input. Otherwise, SPISS is ignored.

Use of FLS Bits in SPI_FLG for Multiple Slave SPI Systems

The FLSx bits in the SPI_FLG register are used in a multiple slave SPI environment. For example, if there are eight SPI devices in the system including a master processor equipped with seven slave selects, the master processor can support the SPI mode transactions across the other seven devices. This configuration requires only one master processor in this multislave environment. For example, assume that the SPI is the master. The seven port pins that can be configured as SPI master mode slave-select output pins can be connected to each of the slave SPI device's SPISS pins. In this configuration, the FLSx bits in SPI_FLG can be used in three cases.

In cases 1 and 2, the processor is the master and the seven microcontrollers/peripherals with SPI interfaces are slaves. The processor can:

1. Transmit to all seven SPI devices at the same time in a broadcast mode. Here, all FLSx bits are set.
2. Receive and transmit from one SPI device by enabling only one slave SPI device at a time.

   In case 3, all eight devices connected through SPI ports can be other processors.
3. If all the slaves are also processors, then the requester can receive data from only one processor (enabled by clearing the EMISO bit in the six other slave processors) at a time and transmit broadcast data to all seven at the same time. This EMISO feature may be available in some other microcontrollers. Therefore, it is possible to use the EMISO feature with any other SPI device that includes this functionality.

Figure 22-4 shows one processor as a master with three processors (or other SPI-compatible devices) as slaves.

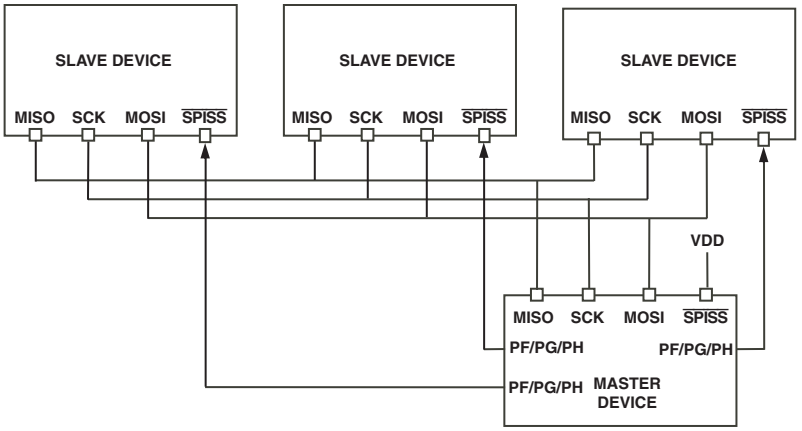


Figure 22-4. Single-Master, Multiple-Slave Configuration

The transmit buffer becomes full after it is written to. It becomes empty when a transfer begins and the transmit value is loaded into the shift register. The receive buffer becomes full at the end of a transfer when the shift register value is loaded into the receive buffer. It becomes empty when the receive buffer is read.

The SPIF bit in the SPI_STAT register is set when the SPI port is disabled.

Upon entering DMA mode, the transmit buffer and the receive buffer become empty. That is, the TXS bit and the RXS bit in the SPI_STAT register are initially cleared upon entering DMA mode.

When using DMA for SPI transmit, the DMA_DONE interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll TXS in the SPI_STAT register until it goes low for two successive reads, at which point the SPI DMA FIFO will be empty. When the SPIF bit subsequently gets set, the last word has been transferred and the SPI can be disabled or enabled for another mode.

Internal Interfaces

The SPI has dedicated connections to the processor's peripheral bus (PAB) and DAB.

The low-latency PAB bus is used to map the SPI resources into the system MMR space. For PAB accesses to SPI MMRs, the primary performance criteria is latency, not throughput. Transfer latencies for both read and write transfers on the peripheral bus are two SCLK cycles.

The DAB bus provides a means for DMA SPI transfers to gain access to on-chip and off-chip memory with little or no degradation in core bandwidth to memory. The SPI peripheral, as a DMA master, is capable of sourcing DMA accesses. The arbitration policy for access to the DAB is described in the *Chip Bus Hierarchy* chapter.

DMA Functionality

The SPI has a single DMA engine which can be configured to support either an SPI transmit channel or a receive channel, but not both simultaneously. Therefore, when configured as a transmit channel, the received data will essentially be ignored.

When configured as a receive channel, what is transmitted is irrelevant. A 16-bit by four-word FIFO (without burst capability) is included to improve throughput on the DAB.

When using DMA for SPI transmit, the DMA_DONE interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll TXS in the SPI_STAT register until it goes low for two successive reads, at which point the SPI DMA FIFO will be empty. When the SPIF bit subsequently gets set, the last word has been transferred and the SPI can be disabled or enabled for another mode.

The four-word FIFO is cleared when the SPI port is disabled.

Description of Operation

The following sections describe the operation of the SPI.

SPI Transfer Protocols

The SPI protocol supports four different combinations of serial clock phase and polarity (SPI modes 0, 1, 2, 3). These combinations are selected using the CPOL and CPHA bits in SPI_CTL as shown in Figure 22-5 on page 22-13.

Figure 22-6 on page 22-14 and Figure 22-7 on page 22-14 demonstrate the two basic transfer formats as defined by the CPHA bit. Two waveforms are shown for SCK—one for CPOL = 0 and the other for CPOL = 1. The diagrams may be interpreted as master or slave timing diagrams since the SCK, MISO, and MOSI pins are directly connected between the master and the slave. The MISO signal is the output from the slave (slave transmission), and the MOSI signal is the output from the master (master transmission). The SCK signal is generated by the master, and the SPISS signal is the slave

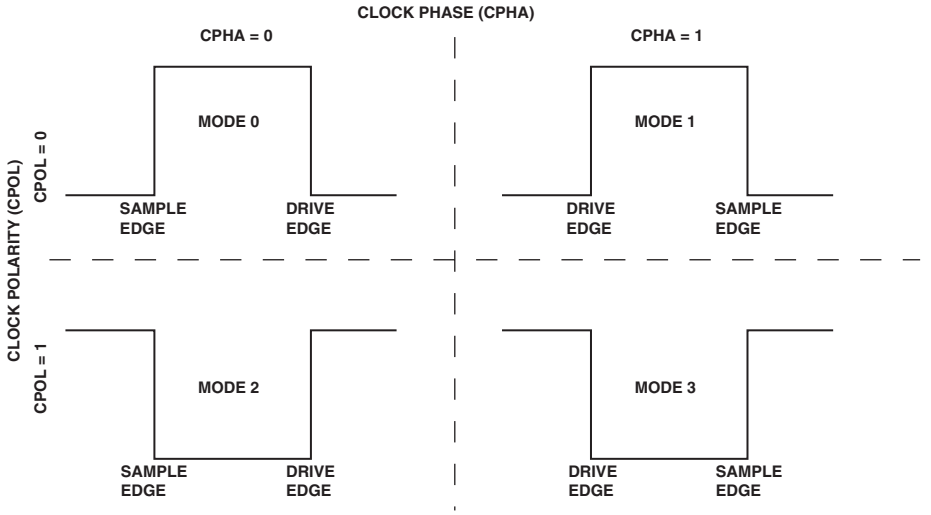


Figure 22-5. SPI Modes of Operation

device select input to the slave from the master. The diagrams represent an 8-bit transfer (SIZE = 0) with the most significant bit (MSB) first (LSBF = 0). Any combination of the SIZE and LSBF bits of SPI_CTL is allowed. For example, a 16-bit transfer with the least significant bit (LSB) first is another possible configuration.

The clock polarity and the clock phase should be identical for the master device and the slave device involved in the communication link. The transfer format from the master may be changed between transfers to adjust to various requirements of a slave device.

When CPHA = 0, the slave select line, SPISS, must be inactive (high) between each serial transfer. This is controlled automatically by the SPI hardware logic. When CPHA = 1, SPISS may either remain active (low) between successive transfers or be inactive (high). This must be controlled by the software through manipulation of the SPI_FLG register.

Figure 22-6 shows the SPI transfer protocol for CPHA = 0. Note SCK starts toggling in the middle of the data transfer, SIZE = 0, and LSBF = 0.

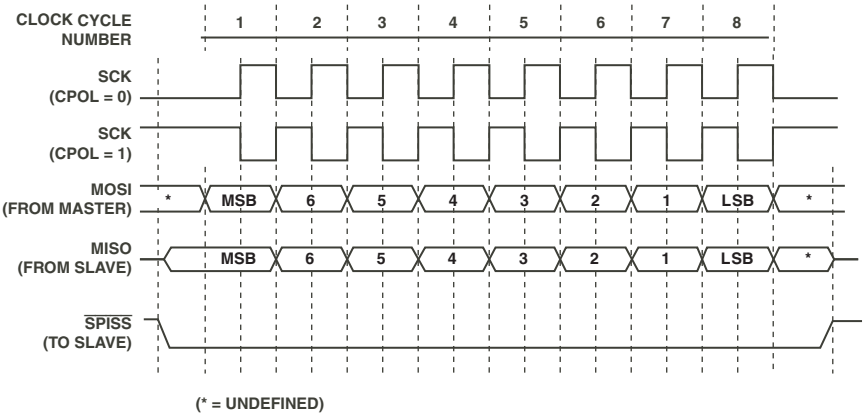


Figure 22-6. SPI Transfer Protocol for CPHA = 0

Figure 22-7 shows the SPI transfer protocol for CPHA = 1. Note SCK starts toggling at the beginning of the data transfer, SIZE = 0, and LSBF = 0.

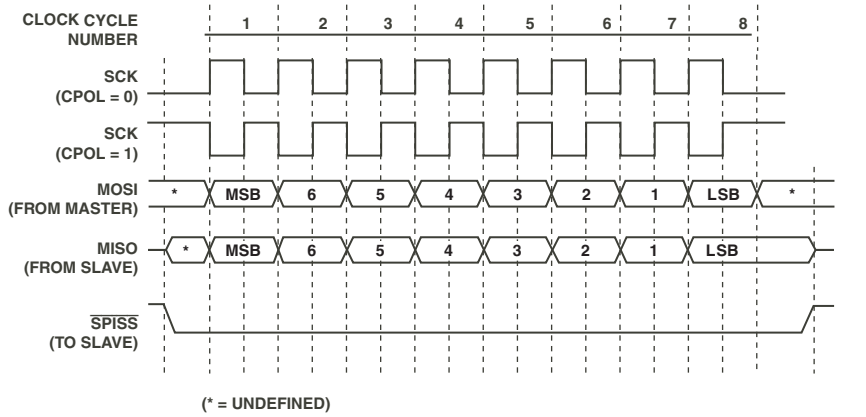


Figure 22-7. SPI Transfer Protocol for CPHA = 1

SPI General Operation

The SPI can be used in single master as well as multimaster environments. The MOSI, MISO, and the SCK signals are all tied together in both configurations. SPI transmission and reception are always enabled simultaneously, unless the broadcast mode has been selected. In broadcast mode, several slaves can be enabled to receive, but only one of the slaves must be in transmit mode driving the MISO line. If the transmit or receive is not needed, it can simply be ignored. This section describes the clock signals, SPI operation as a master and as a slave, and error generation.

Precautions must be taken to avoid data corruption when changing the SPI module configuration. The configuration must not be changed during a data transfer. The clock polarity should only be changed when no slaves are selected. An exception to this is when an SPI communication link consists of a single master and a single slave, CPHA = 1, and the slave select input of the slave is always tied low. In this case, the slave is always selected and data corruption can be avoided by enabling the slave only after both the master and slave devices are configured.

In a multimaster or multislave SPI system, the data output pins (MOSI and MISO) can be configured to behave as open drain outputs, which prevents contention and possible damage to pin drivers. An external pull-up resistor is required on both the MOSI and MISO pins when this option is selected.

The WOM bit in the SPI_CTL register controls this option. When WOM is set and the SPI is configured as a master, the MOSI pin is three-stated when the data driven out on MOSI is a logic high. The MOSI pin is not three-stated when the driven data is a logic low. Similarly, when WOM is set and the SPI is configured as a slave, the MISO pin is three-stated if the data driven out on MISO is a logic high.

During SPI data transfers, one SPI device acts as the SPI link master, where it controls the data flow by generating the SPI serial clock and asserting the SPI device select signal (SPISS). The other SPI device acts as

the slave and accepts new data from the master into its shift register, while it transmits requested data out of the shift register through its SPI transmit data pin. Multiple processors can take turns being the master device, as can other microcontrollers or microprocessors. One master device can also simultaneously shift data into multiple slaves (known as broadcast mode). However, only one slave may drive its output to write data back to the master at any given time. This must be enforced in broadcast mode, where several slaves can be selected to receive data from the master, but only one slave at a time can be enabled to send data back to the master.

In a multimaster or multidevice environment where multiple processors are connected through their SPI ports, all MOSI pins are connected together, all MISO pins are connected together, and all SCK pins are connected together.

For a multislave environment, the processor can make use of up to seven programmable flags that are dedicated SPI slave select signals for the SPI slave devices.

At reset, the SPI is disabled and configured as a slave.

Clock Signals

The SCK signal is a gated clock that is only active during data transfers for the duration of the transferred word. The number of active edges is equal to the number of bits driven on the data lines. The clock rate can be as high as one-fourth of the SCLK rate. For master devices, the clock rate is determined by the 16-bit value in the SPI_BAUD register. For slave devices, the value in SPI_BAUD is ignored. When the SPI device is a master, SCK is an output signal. When the SPI is a slave, SCK is an input signal. Slave devices ignore the serial clock if the slave select input is driven inactive (high).

The SCK signal is used to shift out and shift in the data driven onto the MISO and MOSI lines. The data is always shifted out on one edge of the clock and sampled on the opposite edge of the clock. Clock polarity and

clock phase relative to data are programmable in the SPI_CTL register and define the transfer format. See Figure 22-5 on page 22-13.

Interrupt Output

The SPI has two interrupt output signals: a data interrupt and an error interrupt.

The behavior of the SPI data interrupt signal depends on the TIMOD field in the SPI_CTL register. In DMA mode (TIMOD = b#1X), the data interrupt acts as a DMA request and is generated when the DMA FIFO is ready to be written to (TIMOD = b#11) or read from (TIMOD = b#10). In non-DMA mode (TIMOD = 0X), a data interrupt is generated when the SPI_TDBR register is ready to be written to (TIMOD = b#01) or when the SPI_RDBR register is ready to be read from (TIMOD = b#00).

An SPI error interrupt is generated in a master when a mode fault error occurs, in both DMA and non-DMA modes. An error interrupt can also be generated in DMA mode when there is an underflow (TXE when TIMOD = b#11) or an overflow (RBSY when TIMOD = b#10) error condition. In non-DMA mode, the underflow and overflow conditions set the TXE and RBSY bits in the SPI_STAT register, respectively, but do not generate an error interrupt.

For more information about this interrupt output, see the discussion of the TIMOD bits in "SPI Control (SPI_CTL) Register" on page 22-35.

Functional Description

The following sections describe the functional operation of the SPI.

Master Mode Operation (Non-DMA)

When the SPI is configured as a master (and DMA mode is not selected), the interface operates in the following manner.

1. The core writes to the appropriate port register(s) to properly configure the SPI interface for master mode operation.The required pins are configured for SPI use as slave-select outputs.
2. The core writes to SPI_FLG, setting one or more of the SPI flag select bits (FLSx). This ensures that the desired slaves are properly deselected while the master is configured.
3. The core writes to the SPI_BAUD and SPI_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and other necessary information.
4. If the CPHA bit in the SPI_CTL register = 1, the core activates the desired slaves by clearing one or more of the SPI flag bits (FLGx) of SPI_FLG.
5. The TIMOD bits in SPI_CTL determine the SPI transfer initiate mode. The transfer on the SPI link begins upon either a data write by the core to the SPI_TDBR register or a data read of the SPI_RDBR register.
6. The SPI then generates the programmed clock pulses on SCK and simultaneously shifts data out of MOSI and shifts data in from MISO. Before a shift, the shift register is loaded with the contents of the SPI_TDBR register. At the end of the transfer, the contents of the shift register are loaded into the SPI_RDBR register.
7. With each new transfer initiate command, the SPI continues to send and receive words, according to the SPI transfer initiate mode.

See Figure 22-8 on page 22-30 for additional information.

If the transmit buffer remains empty or the receive buffer remains full, the device operates according to the states of the SZ and GM bits in SPI_CTL.

If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits zeros on the MOSI pin. One word is transmitted for each new transfer initiate command. If SZ = 0 and the transmit buffer is empty, the device repeatedly transmits the last word it transmitted before the transmit buffer became empty.

If GM = 1 and the receive buffer is full, the device continues to receive new data from the MISO pin, overwriting the older data in the SPI_RDBR register. If GM = 0 and the receive buffer is full, the incoming data is discarded, and SPI_RDBR is not updated.

Transfer Initiation From Master (Transfer Modes)

When a device is enabled as a master, the initiation of a transfer is defined by the two TIMOD bits of SPI_CTL. Based on those two bits and the status of the interface, a new transfer is started upon either a read of the SPI_RDBR register or a write to the SPI_TDBR register. This is summarized in Table 22-1.

If the SPI port is enabled with TIMOD = b#01 or TIMOD = b#11, the hardware immediately issues a first interrupt or DMA request.

Table 22-1. Transfer Initiation

TIMOD	Function	Transfer Initiated Upon	Action, Interrupt
b#00	Transmit and receive	Initiate new single word transfer upon read of SPI_RDBR and previous transfer completed.	Interrupt is active when the receive buffer is full. Read of SPI_RDBR clears interrupt.
b#01	Transmit and receive	Initiate new single word transfer upon write to SPI_TDBR and previous transfer completed.	Interrupt is active when the transmit buffer is empty. Writing to SPI_TDBR clears interrupt.
b#10	Receive with DMA	Initiate new multiword transfer upon enabling SPI for DMA mode. Individual word transfers begin with a DMA read of SPI_RDBR, and last transfer completed.	Request DMA reads as long as the SPI DMA FIFO is not empty.
b#11	Transmit with DMA	Initiate new multiword transfer upon enabling SPI for DMA mode. Individual word transfers begin with a DMA write to SPI_TDBR, and last transfer completed.	Request DMA writes as long as the SPI DMA FIFO is not full.

Slave Mode Operation (Non-DMA)

When a device is enabled as a slave (and DMA mode is not selected), the start of a transfer is triggered by a transition of the SPISS select signal to the active state (low), or by the first active edge of the clock (SCK), depending on the state of the CPHA bit in the SPI_CTL register.

These steps illustrate SPI operation in the slave mode:

1. The core writes to the appropriate port register(s) to properly configure the SPI for slave mode operation.
2. The core writes to SPI_CTL to define the mode of the serial link to be the same as the mode set up in the SPI master.
3. To prepare for the data transfer, the core writes data to be transmitted into SPI_TDBR.
4. Once the SPISS falling edge is detected, the slave starts shifting data out on MISO and in from MOSI on SCK edges, depending upon the states of CPHA and CPOL.
5. Reception/transmission continues until SPISS is released or until the slave has received the proper number of clock cycles.
6. The slave device continues to receive/transmit with each new falling edge transition on SPISS and/or SCK clock edge.

See Figure 22-8 on page 22-30 for additional information.

If the transmit buffer remains empty or the receive buffer remains full, the device operates according to the states of the SZ and GM bits in SPI_CTL. If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits zeros on the MISO pin. If SZ = 0 and the transmit buffer is empty, it repeatedly transmits the last word it transmitted before the transmit buffer became empty. If GM = 1 and the receive buffer is full, the device continues to receive new data from the MOSI pin, overwriting the older data in the SPI_RDBR register. If GM = 0 and the receive buffer is full, the incoming data is discarded, and the SPI_RDBR register is not updated.

Slave Ready for a Transfer

When a device is enabled as a slave, the actions shown in Table 22-2 are necessary to prepare the device for a new transfer.

Table 22-2. Transfer Preparation

TIMOD	Function	Action, Interrupt
b#00	Transmit and receive	Interrupt is active when the receive buffer is full. Read of SPI_RDBR clears interrupt.
b#01	Transmit and receive	Interrupt is active when the transmit buffer is empty. Writing to SPI_TDBR clears interrupt.
b#10	Receive with DMA	Request DMA reads as long as SPI DMA FIFO is not empty.
b#11	Transmit with DMA	Request DMA writes as long as SPI DMA FIFO is not full.

Programming Model

The following sections describe the SPI programming model.

Beginning and Ending an SPI Transfer

The start and finish of an SPI transfer depend on whether the device is configured as a master or a slave, which CPHA mode is selected, and which transfer initiation mode (TIMOD) is selected. For a master SPI with CPHA = 0, a transfer starts when either SPI_TDBR is written to or SPI_RDBR is read, depending on TIMOD. At the start of the transfer, the enabled slave select outputs are driven active (low). However, the SCK signal remains inactive for the first half of the first cycle of SCK. For a slave with CPHA = 0, the transfer starts as soon as the SPISS input goes low.

For CPHA = 1, a transfer starts with the first active edge of SCK for both slave and master devices. For a master device, a transfer is considered fin-

ished after it sends the last data and simultaneously receives the last data bit. A transfer for a slave device ends after the last sampling edge of SCK.

The RXS bit defines when the receive buffer can be read. The TXS bit defines when the transmit buffer can be filled. The end of a single word transfer occurs when the RXS bit is set, indicating that a new word has just been received and latched into the receive buffer, SPI_RDBR. For a master SPI, RXS is set shortly after the last sampling edge of SCK. For a slave SPI, RXS is set shortly after the last SCK edge, regardless of CPHA or CPOL. The latency is typically a few SCLK cycles and is independent of TIMOD and the baud rate. If configured to generate an interrupt when SPI_RDBR is full (TIMOD = b#00), the interrupt goes active one SCLK cycle after RXS is set. When not relying on this interrupt, the end of a transfer can be detected by polling the RXS bit.

To maintain software compatibility with other SPI devices, the SPIF bit is also available for polling. This bit may have a slightly different behavior from that of other commercially available devices. For a slave device, SPIF is cleared shortly after the start of a transfer (SPISS going low for CPHA = 0, first active edge of SCK on CPHA = 1), and is set at the same time as RXS. For a master device, SPIF is cleared shortly after the start of a transfer (either by writing the SPI_TDBR or reading the SPI_RDBR, depending on TIMOD), and is set one-half SCK period after the last SCK edge, regardless of CPHA or CPOL.

The time at which SPIF is set depends on the baud rate. In general, SPIF is set after RXS, but at the lowest baud rate settings (SPI_BAUD < 4). The SPIF bit is set before RXS is set, and consequently before new data is latched into SPI_RDBR, because of the latency. Therefore, for SPI_BAUD = 2 or SPI_BAUD = 3, RXS must be set before SPIF to read SPI_RDBR. For larger SPI_BAUD settings, RXS is guaranteed to be set before SPIF is set.

If the SPI port is used to transmit and receive at the same time, or to switch between receive and transmit operation frequently, then the TIMOD = b#00 mode may be the best operation option. In this mode, software performs a dummy read from the SPI_RDBR register to initiate the

first transfer. If the first transfer is used for data transmission, software should write the value to be transmitted into the SPI_TDBR register before performing the dummy read. If the transmitted value is arbitrary, it is good practice to set the SZ bit in the SPI_CTL register to ensure zero data is transmitted rather than random values. When receiving the last word of an SPI stream, software should ensure that the read from the SPI_RDBR register does not initiate another transfer. It is recommended that the SPI port be disabled before the final SPI_RDBR read access. Reading the SPI_SHADOW register is not sufficient, as it does not clear the interrupt request.

In master mode with the CPHA bit set, software should manually assert the required slave select signal before starting the transaction. After all data has been transferred, software typically releases the slave select again. If the SPI slave device requires the slave select line to be asserted for the complete transfer, this can be done in the SPI interrupt service routine only when operating in TIMOD = b#00 or TIMOD = b#10 mode. With TIMOD = b#01 or TIMOD = b#11, the interrupt is requested while the transfer is still in progress.

Master Mode DMA Operation

When enabled as a master with the DMA engine configured to transmit or receive data, the SPI interface operates as follows.

1. The core writes to the appropriate port register(s) to properly configure the SPI for master mode operation. The appropriate pins can be configured for SPI use as slave-select outputs.
2. The processor core writes to the appropriate DMA registers to enable the SPI DMA channel and to configure the necessary work units, access direction, word count, and so on. For more information, see the *Direct Memory Access* chapter.
3. The processor core writes to the SPI_FLG register, setting one or more of the SPI flag select bits (FLSx).

4. The processor core writes to the SPI_BAUD and SPI_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and so on. The TIMOD field should be configured to select either "receive with DMA" (TIMOD = b#10) or "transmit with DMA" (TIMOD = b#11) mode.

5. If configured for receive, a receive transfer is initiated upon enabling of the SPI. Subsequent transfers are initiated as the SPI reads data from the SPI_RDBR register and writes to the SPI DMA FIFO. The SPI then requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

   If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. As the SPI writes data from the SPI DMA FIFO into the SPI_TDBR register, it initiates a transfer on the SPI link.

6. The SPI then generates the programmed clock pulses on SCK and simultaneously shifts data out of MOSI and shifts data in from MISO. For receive transfers, the value in the shift register is loaded into the SPI_RDBR register at the end of the transfer. For transmit transfers, the value in the SPI_TDBR register is loaded into the shift register at the start of the transfer.

7. In receive mode, as long as there is data in the SPI DMA FIFO (the FIFO is not empty), the SPI continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA word count register transitions from "1" to "0". The SPI continues receiving words until SPI DMA mode is disabled.

   In transmit mode, as long as there is room in the SPI DMA FIFO

(the FIFO is not full), the SPI continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI DMA FIFO until the SPI DMA word count register transitions from “1” to “0”. The SPI continues transmitting words until the SPI DMA FIFO is empty.

See Figure 22-9 on page 22-31 for additional information.

For receive DMA operations, if the DMA engine is unable to keep up with the receive datastream, the receive buffer operates according to the state of the GM bit in the SPI_CTL register. If GM = 1 and the DMA FIFO is full, the device continues to receive new data from the MISO pin, overwriting the older data in the SPI_RDBR register. If GM = 0, and the DMA FIFO is full, the incoming data is discarded, and the SPI_RDBR register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty (and TXE is set). If SZ = 1, the device repeatedly transmits zeros on the MOSI pin. If SZ = 0, it repeatedly transmits the contents of the SPI_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, the master SPI initiates a word transfer only when there is data in the DMA FIFO. If the DMA FIFO is empty, the SPI waits for the DMA engine to write to the DMA FIFO before starting the transfer. All aspects of SPI receive operation should be ignored when configured in transmit DMA mode, including the data in the SPI_RDBR register, and the status of the RXS and RBSY bits. The RBSY overrun conditions cannot generate an error interrupt in this mode. The TXE underrun condition cannot happen in this mode (master DMA TX mode), because the master SPI will not initiate a transfer if there is no data in the DMA FIFO.

Writes to the SPI_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the SPI_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPI_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = b#10), or when the DMA FIFO is not full (when TIMOD = b#11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = b#10).

A master SPI DMA sequence may involve back-to-back transmission and/or reception of multiple DMA work units. The SPI controller supports such a sequence with minimal core interaction.

Slave Mode DMA Operation

When enabled as a slave with the DMA engine configured to transmit or receive data, the start of a transfer is triggered by a transition of the SPISS signal to the active-low state or by the first active edge of SCK, depending on the state of CPHA.

The following steps illustrate the SPI receive or transmit DMA sequence in an SPI slave (in response to a master command).

1. The core writes to the appropriate port register(s) to properly configure the SPI for slave mode operation.
2. The processor core writes to the appropriate DMA registers to enable the SPI DMA channel and configure the necessary work units, access direction, word count, and so on. For more information, see the *Direct Memory Access* chapter.
3. The processor core writes to the SPI_CTL register to define the mode of the serial link to be the same as the mode set up in the SPI master. The TIMOD field will be configured to select either “receive with DMA” (TIMOD = b#10) or “transmit with DMA” (TIMOD = b#11) mode.
4. If configured for receive, once the slave select input is active, the slave starts receiving and transmitting data on SCK edges. The value in the shift register is loaded into the SPI_RDBR register at the end

of the transfer. As the SPI reads data from the SPI_RDBR register and writes to the SPI DMA FIFO, it requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. The SPI then reads data from the SPI DMA FIFO and writes to the SPI_TDBR register, awaiting the start of the next transfer. Once the slave select input is active, the slave starts receiving and transmitting data on SCK edges. The value in the SPI_TDBR register is loaded into the shift register at the start of the transfer.

5. In receive mode, as long as there is data in the SPI DMA FIFO (FIFO not empty), the SPI slave continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA word count register transitions from “1” to “0”. The SPI slave continues receiving words on SCK edges as long as the slave select input is active.

   In transmit mode, as long as there is room in the SPI DMA FIFO (FIFO not full), the SPI slave continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI DMA FIFO until the SPI DMA word count register transitions from “1” to “0”. The SPI slave continues transmitting words on SCK edges as long as the slave select input is active.

See Figure 22-9 on page 22-31 for additional information.

For receive DMA operations, if the DMA engine is unable to keep up with the receive datastream, the receive buffer operates according to the state of the GM bit in the SPI_CTL register. If GM = 1 and the DMA FIFO is full, the

device continues to receive new data from the MOSI pin, overwriting the older data in the SPI_RDBR register. If GM = 0 and the DMA FIFO is full, the incoming data is discarded, and the SPI_RDBR register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty and TXE is set. If SZ = 1, the device repeatedly transmits zeros on the MISO pin. If SZ = 0, it repeatedly transmits the contents of the SPI_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, if the DMA engine is unable to keep up with the transmit stream, the transmit port operates according to the state of the SZ bit. If SZ = 1 and the DMA FIFO is empty, the device repeatedly transmits zeros on the MISO pin. If SZ = 0 and the DMA FIFO is empty, it repeatedly transmits the last word it transmitted before the DMA buffer became empty. All aspects of SPI receive operation should be ignored when configured in transmit DMA mode, including the data in the SPI_RDBR register, and the status of the RXS and RBSY bits. The RBSY overrun conditions cannot generate an error interrupt in this mode.

Writes to the SPI_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the SPI_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPI_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = b#10), or when the DMA FIFO is not full (when TIMOD = b#11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = b#10), or when there is a TXE underflow error condition (when TIMOD = b#11).

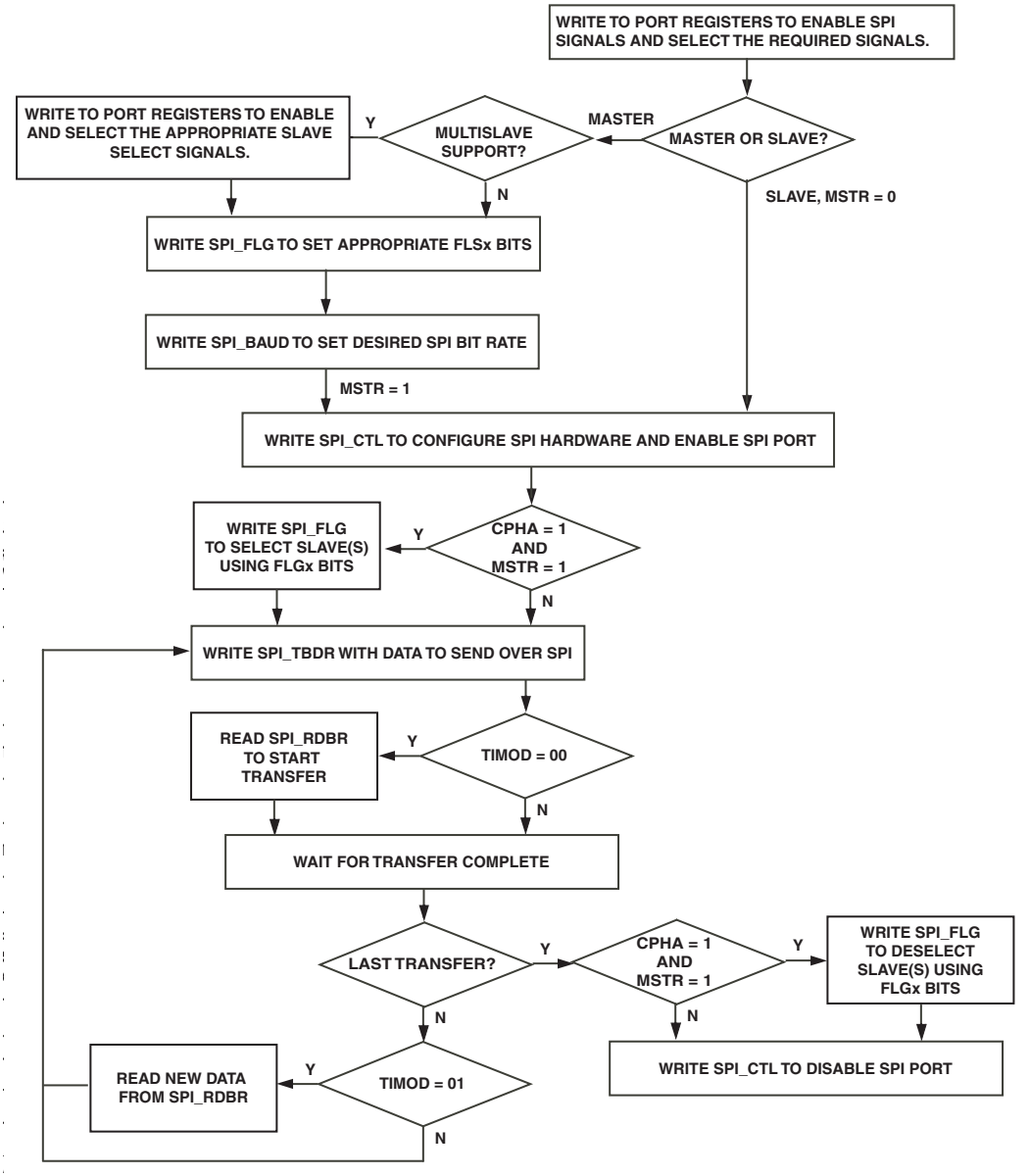

WRITE TO PORT REGISTERS TO ENABLE SPI SIGNALS AND SELECT THE REQUIRED SIGNALS.
MASTER OR SLAVE?
MASTER
SLAVE, MSTR = 0
MULTISLAVE SUPPORT?
Y
N
WRITE TO PORT REGISTERS TO ENABLE AND SELECT THE APPROPRIATE SLAVE SELECT SIGNALS.
WRITE SPI_FLG TO SET APPROPRIATE FLSx BITS
WRITE SPI_BAUD TO SET DESIRED SPI BIT RATE
MSTR = 1
WRITE SPI_CTL TO CONFIGURE SPI HARDWARE AND ENABLE SPI PORT
CPHA = 1 AND MSTR = 1
Y
N
WRITE SPI_FLG TO SELECT SLAVE(S) USING FLGx BITS
WRITE SPI_TBDR WITH DATA TO SEND OVER SPI
TIMOD = 00
Y
N
READ SPI_RDBR TO START TRANSFER
WAIT FOR TRANSFER COMPLETE
LAST TRANSFER?
Y
N
CPHA = 1 AND MSTR = 1
Y
N
WRITE SPI_FLG TO DESELECT SLAVE(S) USING FLGx BITS
WRITE SPI_CTL TO DISABLE SPI PORT
TIMOD = 01
Y
N
READ NEW DATA FROM SPI_RDBR

Figure 22-8. Core-Driven SPI Flow Chart

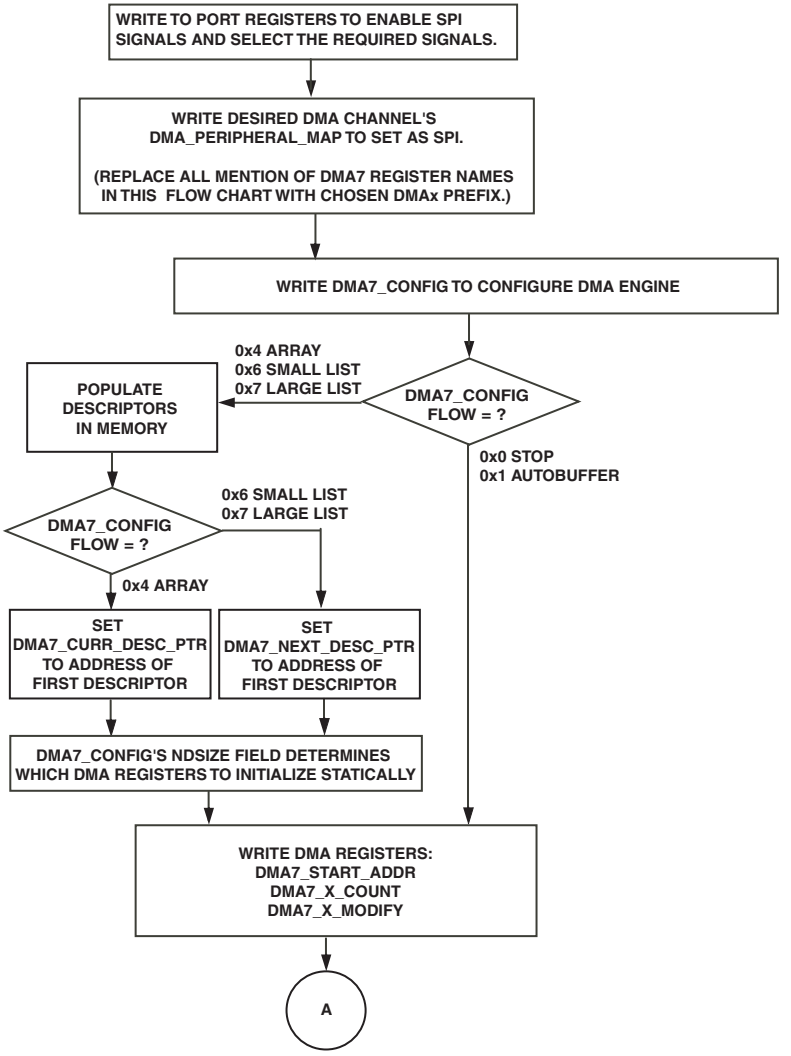

WRITE TO PORT REGISTERS TO ENABLE SPI SIGNALS AND SELECT THE REQUIRED SIGNALS.
WRITE DESIRED DMA CHANNEL'S DMA_PERIPHERAL_MAP TO SET AS SPI.
(REPLACE ALL MENTION OF DMA7 REGISTER NAMES IN THIS FLOW CHART WITH CHOSEN DMAx PREFIX.)
WRITE DMA7_CONFIG TO CONFIGURE DMA ENGINE
0x4 ARRAY
0x6 SMALL LIST
0x7 LARGE LIST
DMA7_CONFIG FLOW = ?
POPULATE DESCRIPTORS IN MEMORY
0x0 STOP
0x1 AUTOBUFFER
0x6 SMALL LIST
0x7 LARGE LIST
DMA7_CONFIG FLOW = ?
0x4 ARRAY
SET DMA7_CURR_DESC_PTR TO ADDRESS OF FIRST DESCRIPTOR
SET DMA7_NEXT_DESC_PTR TO ADDRESS OF FIRST DESCRIPTOR
DMA7_CONFIG'S NDSIZE FIELD DETERMINES WHICH DMA REGISTERS TO INITIALIZE STATICALLY
WRITE DMA REGISTERS:
DMA7_START_ADDR
DMA7_X_COUNT
DMA7_X_MODIFY
A

Figure 22-9. SPI DMA Flow Chart (Part 1 of 3)

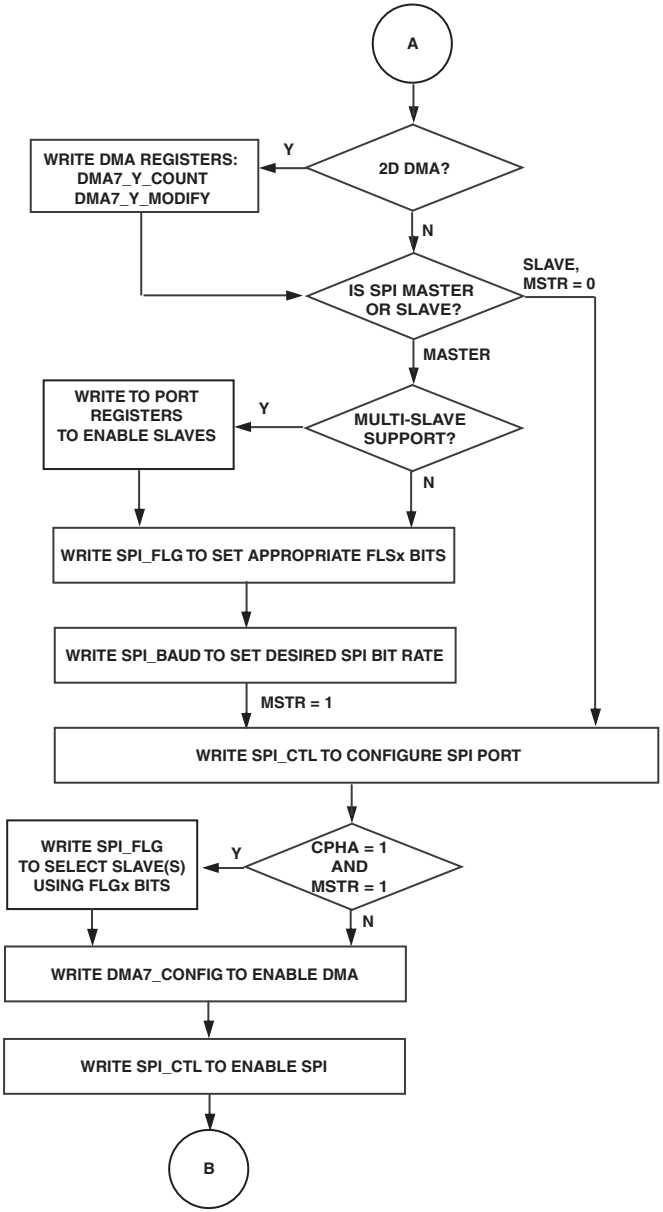


Figure 22-10. SPI DMA Flow Chart (Part 2 of 3)

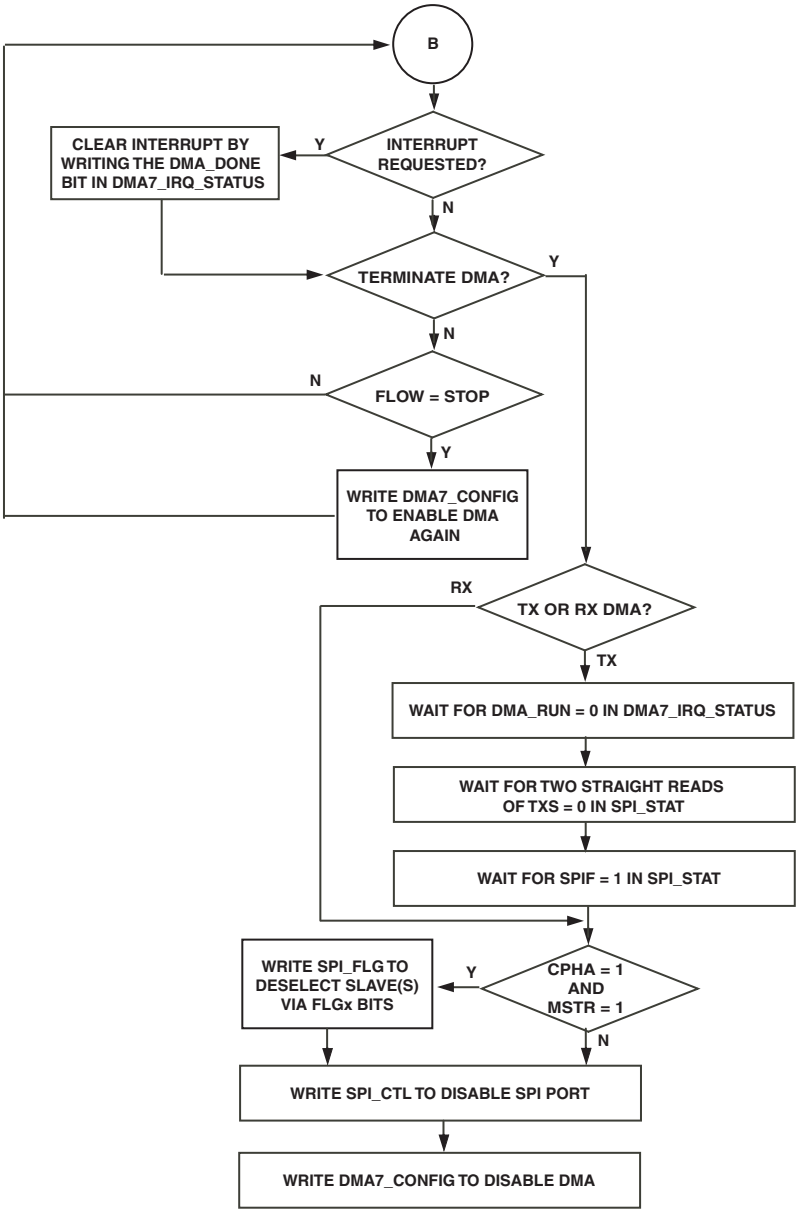


Figure 22-11. SPI DMA Flow Chart (Part 3 of 3)

SPI Registers

The SPI peripheral includes a number of user-accessible registers. Some of these registers are also accessible through the DMA bus. Four registers contain control and status information: SPI_BAUD, SPI_CTL, SPI_FLG, and SPI_STAT. Two registers are used for buffering receive and transmit data: SPI_RDBR and SPI_TDBR. The shift register, SFDR, is internal to the SPI module and is not directly accessible.

Table 22-3shows the functions of the SPI registers. Figure 22-12 through Figure 22-18 on page 22-44 provide details.

Table 22-3. SPI Register Mapping

Register Name	Function	Notes
SPI_BAUD	SPI port baud control	Value of “0” or “1” disables the serial clock
SPI_CTL	SPI port control	SPE and MSTR bits can also be modified by hardware (when MODF is set)
SPI_FLG	SPI port flag	Bits 0 and 8 are reserved
SPI_STAT	SPI port status	SPIF bit can be set by clearing SPE in SPI_CTL
SPI_TDBR	SPI port transmit data buffer	Register contents can also be modified by hardware (by DMA and/or when SZ = 1 in SPI_CTL)
SPI_RDBR	SPI port receive data buffer	When register is read, hardware events can be triggered
SPI_SHADOW	SPI port data	Register has the same contents as SPI_RDBR, but no action is taken when it is read

SPI Baud Rate (SPI_BAUD) Register

The SPI_BAUD register is used to set the bit transfer rate for a master device. When configured as a slave, the value written to this register is ignored. The serial clock frequency is determined by this formula:

SCK frequency = (peripheral clock frequency SCLK)/(2 × SPI_BAUD)

Writing a value of “0” or “1” to the register disables the serial clock. Therefore, the maximum serial clock rate is one-fourth the system clock rate.

Table 22-4 lists several possible baud rate values for SPI_BAUD.

Table 22-4. SPI Master Baud Rate Example

SPI_BAUD Decimal Value	SPI Clock (SCK) Divide Factor	Baud Rate for SCLK at 100 MHz
0	N/A	N/A
1	N/A	N/A
2	4	25 MHz
3	6	16.7 MHz
4	8	12.5 MHz
65,535 (0xFFFF)	131,070	763 Hz

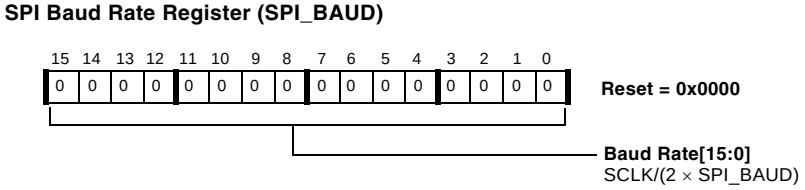


Figure 22-12. SPI Baud Rate Register

SPI Control (SPI_CTL) Register

The SPI_CTL register is used to configure and enable the SPI system. This register is used to enable the SPI interface, select the device as a master or slave, and determine the data transfer format and word size.

The term “word” refers to a single data transfer of either 8 bits or 16 bits, depending on the word length (SIZE) bit in SPI_CTL. There are two special bits which can also be modified by the hardware: SPE and MSTR.

The TIMOD field is used to specify the action that initiates transfers to/from the receive/transmit buffers. When set to b#00, a SPI port transaction is begun when the receive buffer is read. Data from the first read will need to be discarded since the read is needed to initiate the first SPI port transaction. When set to b#01, the transaction is initiated when the transmit buffer is written. A value of b#10 selects DMA receive mode and the first transaction is initiated by enabling the SPI for DMA receive mode. Subsequent individual transactions are initiated by a DMA read of the SPI_RDBR register. A value of 11 selects DMA transmit mode and the transaction is initiated by a DMA write of the SPI_TDBR register.

The PSSE bit is used to enable the SPISS input for an external master. When not used, SPISS can be disabled, freeing up a pin for an alternate function.

The EMISO bit enables the MISO pin as an output. This is needed in an environment where the master wishes to transmit to various slaves at one time (broadcast). Only one slave is allowed to transmit data back to the master. Except for the slave from whom the master wishes to receive, all other slaves should have this bit cleared.

The SPE and MSTR bits can be modified by hardware when the MODF bit of the SPI_STAT register is set. See “Mode Fault Error (MODF)” on page 22-41.

Figure 22-13 on page 22-37 provides the bit descriptions for SPI_CTL.

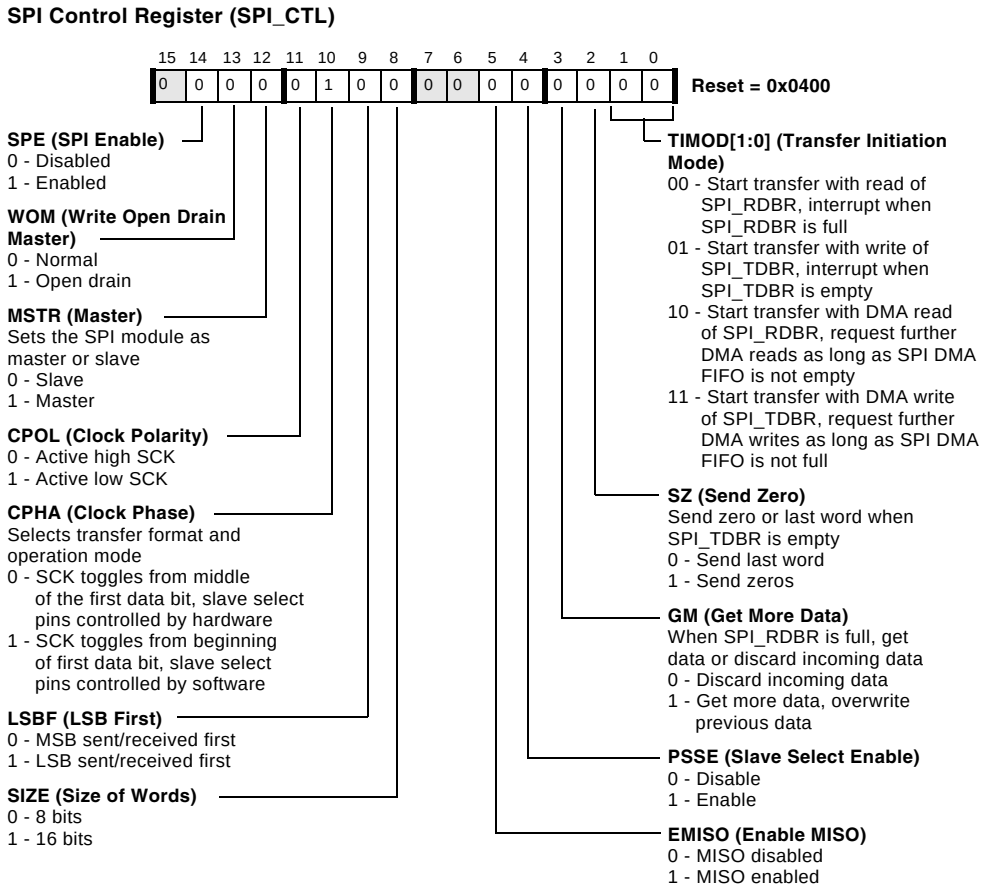

SPI Control Register (SPI_CTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
Reset = 0x0400
SPE (SPI Enable)
0 - Disabled
1 - Enabled
WOM (Write Open Drain Master)
0 - Normal
1 - Open drain
MSTR (Master)
Sets the SPI module as master or slave
0 - Slave
1 - Master
CPOL (Clock Polarity)
0 - Active high SCK
1 - Active low SCK
CPHA (Clock Phase)
Selects transfer format and operation mode
0 - SCK toggles from middle of the first data bit, slave select pins controlled by hardware
1 - SCK toggles from beginning of first data bit, slave select pins controlled by software
LSBF (LSB First)
0 - MSB sent/received first
1 - LSB sent/received first
SIZE (Size of Words)
0 - 8 bits
1 - 16 bits
TIMOD[1:0] (Transfer Initiation Mode)
00 - Start transfer with read of SPI_RDBR, interrupt when SPI_RDBR is full
01 - Start transfer with write of SPI_TDBR, interrupt when SPI_TDBR is empty
10 - Start transfer with DMA read of SPI_RDBR, request further DMA reads as long as SPI DMA FIFO is not empty
11 - Start transfer with DMA write of SPI_TDBR, request further DMA writes as long as SPI DMA FIFO is not full
SZ (Send Zero)
Send zero or last word when SPI_TDBR is empty
0 - Send last word
1 - Send zeros
GM (Get More Data)
When SPI_RDBR is full, get data or discard incoming data
0 - Discard incoming data
1 - Get more data, overwrite previous data
PSSE (Slave Select Enable)
0 - Disable
1 - Enable
EMISO (Enable MISO)
0 - MISO disabled
1 - MISO enabled

Figure 22-13. SPI Control Register

SPI Flag (SPI_FLG) Register

The SPI_FLG register consists of two sets of bits that function as follows.

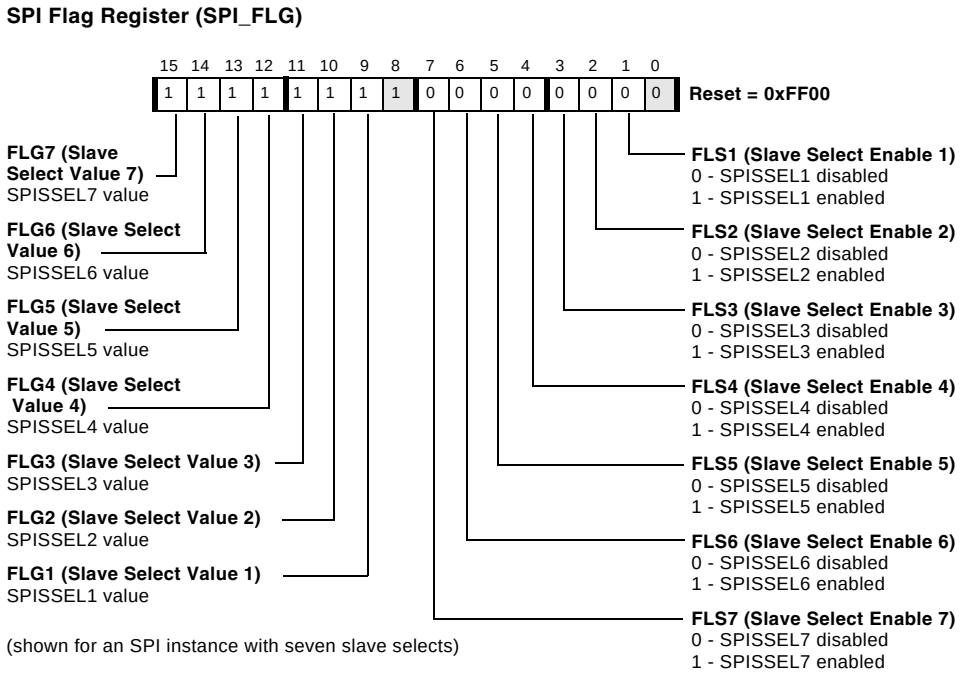

SPI Flag Register (SPI_FLG)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
Reset = 0xFF00
FLG7 (Slave Select Value 7)
SPISSEL7 value
FLG6 (Slave Select Value 6)
SPISSEL6 value
FLG5 (Slave Select Value 5)
SPISSEL5 value
FLG4 (Slave Select Value 4)
SPISSEL4 value
FLG3 (Slave Select Value 3)
SPISSEL3 value
FLG2 (Slave Select Value 2)
SPISSEL2 value
FLG1 (Slave Select Value 1)
SPISSEL1 value
(shown for an SPI instance with seven slave selects)
FLS1 (Slave Select Enable 1)
0 - SPISSEL1 disabled
1 - SPISSEL1 enabled
FLS2 (Slave Select Enable 2)
0 - SPISSEL2 disabled
1 - SPISSEL2 enabled
FLS3 (Slave Select Enable 3)
0 - SPISSEL3 disabled
1 - SPISSEL3 enabled
FLS4 (Slave Select Enable 4)
0 - SPISSEL4 disabled
1 - SPISSEL4 enabled
FLS5 (Slave Select Enable 5)
0 - SPISSEL5 disabled
1 - SPISSEL5 enabled
FLS6 (Slave Select Enable 6)
0 - SPISSEL6 disabled
1 - SPISSEL6 enabled
FLS7 (Slave Select Enable 7)
0 - SPISSEL7 disabled
1 - SPISSEL7 enabled

Figure 22-14. SPI Flag Register (example with 7 slave selects)

- Slave select enable (FLSx) bits

  Each FLSx bit corresponds to a general purpose port pin. When an FLSx bit is set, the corresponding port pin is driven as a slave select. For example, if FLS1 is set in SPI_FLG, the port pin corresponding to SPISSEL1 is driven as a slave select.

If the FLSx bit is not set, the general-purpose port registers configure and control the corresponding port pins.

- Slave select value (FLGx) bits
- When a port pin is configured as a slave select output, the FLGx bits can determine the value driven onto the output. If the CPHA bit in SPI_CTL is set, the output value is set by software control of the FLGx bits. The SPI protocol permits the slave select line to either remain asserted (low) or be deasserted between transferred words. The user must set or clear the appropriate FLGx bits. For example, setting FLS3 in the SPI_FLG register drives the SPISSEL3 pin as a slave select. Then, clearing FLG3 in the SPI_FLG register drives the pin low, and setting FLG3 drives it high. The pin can be cycled high and low between transfers by setting and clearing FLG3. Otherwise, the pin remains active (low) between transfers.

  If CPHA = 0, the SPI hardware sets the output value and the FLGx bits are ignored. The SPI protocol requires that the slave select be deasserted between transferred words. In this case, the SPI hardware controls the pins. For example, to use the slave select function on a port pin to which it is mapped, it is only necessary to set the appropriate FLS bit in SPI_FLG. It is not necessary to write to an FLG bit, because the SPI hardware automatically drives the port pin.

SPI Status (SPI_STAT) Register

The SPI_STAT register is used to detect when an SPI transfer is complete or if transmission/reception errors occur. The SPI_STAT register can be read at any time.

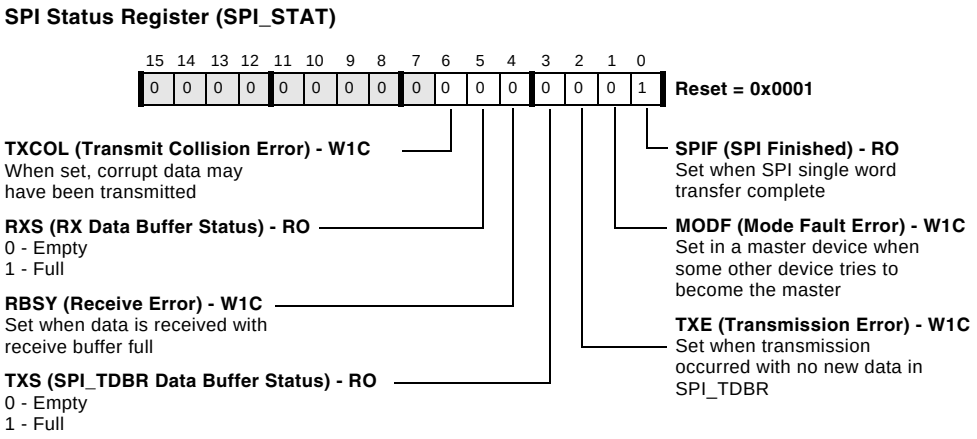

SPI Status Register (SPI_STAT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
Reset = 0x0001
TXCOL (Transmit Collision Error) - W1C
When set, corrupt data may have been transmitted
RXS (RX Data Buffer Status) - RO
0 - Empty
1 - Full
RBSY (Receive Error) - W1C
Set when data is received with receive buffer full
TXS (SPI_TDBR Data Buffer Status) - RO
0 - Empty
1 - Full
SPIF (SPI Finished) - RO
Set when SPI single word transfer complete
MODF (Mode Fault Error) - W1C
Set in a master device when some other device tries to become the master
TXE (Transmission Error) - W1C
Set when transmission occurred with no new data in SPI_TDBR

Figure 22-15. SPI Status Register

Some of the bits in SPI_STAT are read-only and other bits are sticky. Bits that provide information only about the SPI are read-only. These bits are set and cleared by the hardware. Sticky bits are set when an error condition occurs. These bits are set by hardware and must be cleared by software. To clear a sticky bit, the user must write a "1" to the desired bit position of SPI_STAT. For example, if the TXE bit is set, the user must write a "1" to bit 2 of SPI_STAT to clear the TXE error condition. This allows the user to read SPI_STAT without changing its value.

Sticky bits are cleared on a reset, but are not cleared on an SPI disable.

See Figure 22-15 on page 22-40 for more information.

Mode Fault Error (MODF)

The MODF bit is set in SPI_STAT when the SPISS input pin of a device enabled as a master is driven low by some other device in the system. This occurs in multimaster systems when another device is also trying to be the master. To enable this feature, the PSSE bit in SPI_CTL must be set. This contention between two drivers can potentially damage the driving pins. As soon as this error is detected, these actions occur:

- The MSTR control bit in SPI_CTL is cleared, configuring the SPI interface as a slave
- The SPE control bit in SPI_CTL is cleared, disabling the SPI system
- The MODF status bit in SPI_STAT is set
- An SPI error interrupt is generated

These four conditions persist until the MODF bit is cleared by software. Until the MODF bit is cleared, the SPI cannot be re-enabled, even as a slave. Hardware prevents the user from setting either SPE or MSTR while MODF is set.

When MODF is cleared, the interrupt is deactivated. Before attempting to re-enable the SPI as a master, the state of the SPISS input pin should be checked to make sure the pin is high. Otherwise, once SPE and MSTR are set, another mode fault error condition immediately occurs.

When SPE and MSTR are cleared, the SPI data and clock pin drivers (MOSI, MISO, and SCK) are disabled. However, the slave select output pins revert to being controlled by the general-purpose I/O port registers. This could lead to contention on the slave select lines if these lines are still driven by the processor. To ensure that the slave select output drivers are disabled once an MODF error occurs, the program must configure the general-purpose I/O port registers appropriately.

When enabling the MODF feature, the program must configure as inputs all of the port pins that will be used as slave selects. Programs can do this by

configuring the direction of the port pins prior to configuring the SPI. This ensures that, once the MODF error occurs and the slave selects are automatically reconfigured as port pins, the slave select output drivers are disabled.

Transmission Error (TXE)

The TXE bit is set in SPI_STAT when all the conditions of transmission are met, and there is no new data in SPI_TDBR (SPI_TDBR is empty). In this case, the contents of the transmission depend on the state of the SZ bit in SPI_CTL. The TXE bit is sticky (W1C).

Reception Error (RBSY)

The RBSY flag is set in the SPI_STAT register when a new transfer is completed, but before the previous data can be read from SPI_RDBR. The state of the GM bit in the SPI_CTL register determines whether SPI_RDBR is updated with the newly received data. The RBSY bit is sticky (W1C).

Transmit Collision Error (TXCOL)

The TXCOL flag is set in SPI_STAT when a write to SPI_TDBR coincides with the load of the shift register. The write to SPI_TDBR can be by software or the DMA. The TXCOL bit indicates that corrupt data may have been loaded into the shift register and transmitted. In this case, the data in SPI_TDBR may not match what was transmitted. This error can easily be avoided by proper software control. The TXCOL bit is sticky (W1C).

SPI Transmit Data Buffer (SPI_TDBR) Register

The SPI_TDBR register is a 16-bit read-write register. Data is loaded into this register before being transmitted. Just prior to the beginning of a data transfer, the data in SPI_TDBR is loaded into the internal shift register SFDR. A read of SPI_TDBR can occur at any time and does not interfere with or initiate SPI transfers.

When the DMA is enabled for transmit operation, the DMA engine loads data into this register for transmission just prior to the beginning of a data transfer. A write to SPI_TDBR should not occur in this mode because this data will overwrite the DMA data to be transmitted.

When the DMA is enabled for receive operation, the contents of SPI_TDBR are repeatedly transmitted. A write to SPI_TDBR is permitted in this mode, and this data is transmitted.

If the SZ control bit in the SPI_CTL register is set, SPI_TDBR may be reset to zero under certain circumstances.

If multiple writes to SPI_TDBR occur while a transfer is already in progress, only the last data written is transmitted. None of the intermediate values written to SPI_TDBR are transmitted. Multiple writes to SPI_TDBR are possible, but not recommended.

SPI Transmit Data Buffer Register (SPI_TDBR)

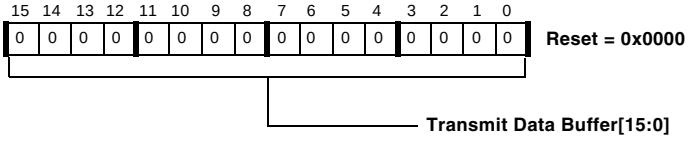


Figure 22-16. SPI Transmit Data Buffer Register

SPI Receive Data Buffer (SPI_RDBR) Register

The SPI_RDBR register is a 16-bit read-only register. At the end of a data transfer, the data in the shift register is loaded into SPI_RDBR. During a DMA receive operation, the data in SPI_RDBR is automatically read by the DMA controller. When SPI_RDBR is read by software, the RXS bit in the

SPI_STAT register is cleared and an SPI transfer may be initiated (if TIMOD = b#00).

SPI Receive Data Buffer Register (SPI_RDBR)
Read Only

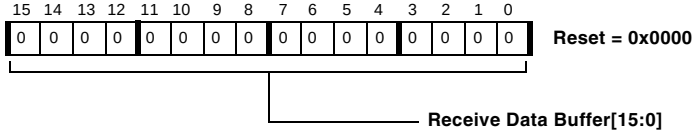


Figure 22-17. SPI Receive Data Buffer Register

SPI RDBR Shadow (SPI_SHADOW) Register

The SPI_SHADOW register is provided for use in debugging software. This register is at a different address than the receive data buffer, SPI_RDBR, but its contents are identical to that of SPI_RDBR. When a software read of SPI_RDBR occurs, the RXS bit in SPI_STAT is cleared and an SPI transfer may be initiated (if TIMOD = b#00 in SPI_CTL). No such hardware action occurs when the SPI_SHADOW register is read. The SPI_SHADOW register is read-only.

SPI RDBR Shadow Register (SPI_SHADOW)
Read Only

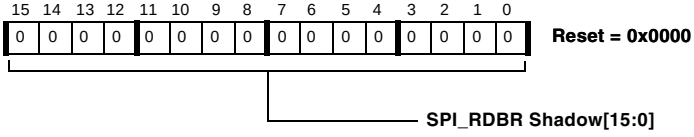


Figure 22-18. SPI RDBR Shadow Register

Programming Examples

This section includes examples (Listing 22-1 through Listing 22-8 on page 22-52) for both core-generated and DMA-based transfers. Each code example assumes that the appropriate processor header files are included.

Core-Generated Transfer

The following core-driven master-mode SPI example shows how to initialize the hardware, signal the start of a transfer, handle the interrupt and issue the next transfer, and generate a stop condition.

Initialization Sequence

Before the SPI can transfer data, the registers must be configured as follows.

Listing 22-1. SPI Register Initialization

```
SPI_Register_Initialization:
  P0.H = hi(SPI_FLG);
  P0.L = lo(SPI_FLG);
  R0 = W[P0] (Z);
  BITSET (R0,0x7);      /* FLS7 */
  W[P0] = R0;           /* Enable slave-select output pin */

  P0.H = hi(SPI_BAUD);
  P0.L = lo(SPI_BAUD);
  R0.L = 0x208E;        /* Write to SPI Baud rate register */
  W[P0] = R0.L; ssync; /*If SCLK = 133 MHz, SPI clock ~= 8 kHz*/

/* Setup SPI Control Register */
/***********************************************/
 /* TIMOD [1:0] = 00 : Transfer On RDBR Read.*/
```

```
 /* SZ [2]      = 0 : Send Last Word When TDBR Is Empty*/
 /* GM [3]      = 1 : Overwrite Previous Data If RDBR Is Full*/
 /* PSSE [4]    = 0 : Disables Slave-Select As Input (Master)*/
 /* EMISO [5]   = 0 : MISO Disabled For Output (Master)*/
 /* [7] and [6] = 0 : RESERVED*/
 /* SIZE [8]    = 1 : 16 Bit Word Length Select*/
 /* LSBF [9]    = 0 : Transmit MSB First*/
 /* CPHA [10]   = 0 : Hardware Controls Slave-Select Outputs*/
 /* CPOL [11]   = 1 : Active Low SCK*/
 /* MSTR [12]   = 1 : Device Is Master*/
 /* WOM [13]    = 0 : Normal MOSI/MISO Data Output (No Open
Drain)*/
 /* SPE [14]    = 1 : SPI Module Is Enabled*/
 /* [15]        = 0 : RESERVED*/
 /*********************************************/
  P0.H = hi(SPI_CTL) ;
  P0.L = lo(SPI_CTL) ;
  R0 = 0x5908;
  W[P0] = R0.L; ssync;   /* Enable SPI as MASTER */
```

Starting a Transfer

After the initialization procedure in the given master mode, a transfer begins following a dummy read of `SPI_RDBR`. Typically, known data which is desired to be transmitted to the slave is preloaded into the `SPI_TDBR`. In the following code, `P1` is assumed to point to the start of the 16-bit transmit data buffer and `P2` is assumed to point to the start of the 16-bit receive data buffer. In addition, the user must ensure appropriate interrupts are enabled for SPI operation.

Listing 22-2. Initiate Transfer

```
Initiate_Transfer:
  P0.H = hi(SPI_FLG);
```

```
P0.L = lo(SPI_FLG);
R0 = W[P0] (Z);
BITCLR (R0,0xF);      /* FLG7 */
W[P0] = R0;           /* Drive 0 on enabled slave-select pin */

P0.H = hi(SPI_TDBR);  /* SPI Transmit Register */
P0.L = lo(SPI_TDBR);
R0 = W[P1++] (z);     /* Get First Data To Be Transmitted
And Increment Pointer */
W[P0] = R0;            /* Write to SPI_TDBR */

P0.H = hi(SPI_RDBR);
P0.L = lo(SPI_RDBR);
R0 = W[P0] (z); /* Dummy read of SPI_RDBR kicks off transfer */
```

Post Transfer and Next Transfer

Following the transfer of data, the SPI generates an interrupt, which is serviced if the interrupt is enabled during initialization. In the interrupt routine, software must write the next value to be transmitted prior to reading the byte received. This is because a read of the `SPI_RDBR` initiates the next transfer.

Listing 22-3. SPI Interrupt Handler

```
SPI_Interrupt_Handler
Process_SPI_Sample:
  P0.H = hi(SPI_TDBR);     /* SPI transmit register */
  P0.L = lo(SPI_TDBR);
  R0 = W[P1++](z);         /* Get next data to be transmitted */
  W[P0] = R0.l;            /* Write that data to SPI_TDBR */

Kick_Off_Next:
  P0.H = hi(SPI_RDBR);     /* SPI receive register */
```

```
  P0.L = lo(SPI_RDBR);
  R0 = W[P0] (z);   /* Read SPI receive register (also kicks off
next transfer) */
  W[P2++] = R0;            /* Store received data to memory */
  RTI;                     /* Exit interrupt handler */
```

Stopping

In order for a data transfer to end after the user has transferred all data, the following code can be used to stop the SPI. Note that this is typically done in the interrupt handler to ensure the final data has been sent in its entirety.

Listing 22-4. Stopping SPI

```
Stopping_SPI:
  P0.H = hi(SPI_CTL);
 P0.L = lo(SPI_CTL);
  R0 = W[P0];
  BITCLR(R0, 14);        /* Clear SPI enable bit */
  W[P0] = R0.L; ssync;   /* Disable SPI */
```

DMA-Based Transfer

The following DMA-driven master-mode SPI autobuffer example shows how to initialize DMA, initialize SPI, signal the start of a transfer, and generate a stop condition.

DMA Initialization Sequence

The following code initializes the DMA to perform a 16-bit memory read DMA operation in autobuffer mode, and generates an interrupt request when the buffer has been sent. This code assumes that `P1` points to the start of the data buffer to be transmitted and that `NUM_SAMPLES` is a defined macro indicating the number of elements being sent.

Listing 22-5. DMA Initialization

```
Initialize_DMA:    /* Assume DMA7 as the channel for SPI DMA */
    P0.H = hi(DMA7_CONFIG);
    P0.L = lo(DMA7_CONFIG);
    R0 = 0x1084(z);   /* Autobuffer mode, IRQ on complete, linear
16-bit, mem read */
    w[P0] = R0;
  P0.H = hi(DMA7_START_ADDR);
  P0.L = lo(DMA7_START_ADDR);
  [p0] = p1;     /* Start address of TX buffer */  P0.H  =
hi(DMA7_X_COUNT);
  P0.L  = lo(DMA7_X_COUNT);
  R0    = NUM_SAMPLES;
  w[p0] = R0;   /* Number of samples to transfer */
  R0 = 2;
  P0.H = hi(DMA7_X_MODIFY);
  P0.L = lo(DMA7_X_MODIFY);
  w[p0] = R0;   /* 2 byte stride for 16-bit words */
  R0 = 1;       /* single dimension DMA means 1 row */
  P0.H = hi(DMA7_Y_COUNT);
  P0.L = lo(DMA7_Y_COUNT);
  w[p0] = R0;
```

SPI Initialization Sequence

Before the SPI can transfer data, the registers must be configured as follows.

Listing 22-6. SPI Initialization

```
SPI_Register_Initialization:
  P0.H = hi(SPI_FLG);
  P0.L = lo(SPI_FLG);
```

```
  R0 = W[P0] (Z);
  BITSET (R0,0x7);    /* FLS7 */
  W[P0] = R0;         /* Enable slave-select output pin */
  P1.H = hi(SPI_BAUD);
  P1.L = lo(SPI_BAUD);
  R0.L = 0x208E;      /* Write to SPI baud rate register */
  W[P0] = R0.L; ssync;  /* If SCLK = 133MHz, SPI clock ~= 8kHz */
  /* Setup SPI Control Register */
/************************************************
 * TIMOD [1:0] = 11 : Transfer on DMA TDBR write
 * SZ [2]      =  0 : Send last word when TDBR is empty
 * GM [3]      =  1 : Discard incoming data if RDBR is full
 * PSSE [4]    =  0 : Disables slave-select as input (master)
 * EMISO [5]   =  0 : MISO disabled for output (master)
 * [7] and [6] =  0 : RESERVED
 * SIZE [8]    =  1 : 16 Bit word length select
 * LSBF [9]    =  0 : Transmit MSB first
 * CPHA [10]   =  0 : SCK starts toggling at START of first data
bit
 * CPOL [11]   =  1 : Active HIGH serial clock
 * MSTR [12]   =  1 : Device is master
 * WOM [13]    =  0 : Normal MOSI/MISO data output (no open
drain)
 * SPE [14]    =  1 : SPI module is enabled
 * [15]        =  0 : RESERVED
 ************************************************/
  /* Configure SPI as MASTER */
  R1 = 0x190B(z);     /* Leave disabled until DMA is enabled */
  P1.L = lo(SPI_CTL);
  W[P1] = R1; ssync;
```

Starting a Transfer

After the initialization procedure in the given master mode, a transfer begins following enabling of SPI. However, the DMA must be enabled before enabling the SPI.

Listing 22-7. Starting a Transfer

```
Initiate_Transfer:
  P0.H = hi(DMA7_CONFIG);
  P0.L = lo(DMA7_CONFIG);
  R2 = w[P0](z);
  BITSET (R2, 0);   /*Set DMA enable bit */
  w[p0] = R2.L;     /* Enable TX DMA */

  P4.H = hi(SPI_CTL);
  P4.L = lo(SPI_CTL);
  R2=w[p4](z);
  BITSET (R2, 14);  /* Set SPI enable bit */
  w[p4] = R2;       /* Enable SPI */
```

Stopping a Transfer

In order for a data transfer to end after the DMA has transferred all required data, the following code is executed in the SPI DMA interrupt handler. The example code below clears the DMA interrupt, then waits for the DMA engine to stop running. When the DMA engine has completed, `SPI_STAT` is polled to determine when the transmit buffer is empty. If there is data in the SPI Transmit FIFO, it is loaded as soon as the `TXS` bit clears. A second consecutive read with the `TXS` bit clear indicates the FIFO is empty and the last word is in the shift register. Finally, polling for the `SPIF` bit determines when the last bit of the last word has been shifted out. At that point, it is safe to shut down the SPI port and the DMA engine.

Listing 22-8. Stopping a Transfer

```
SPI_DMA_INTERRUPT_HANDLER:
  P0.L = lo(DMA7_IRQ_STATUS);
  P0.H = hi(DMA7_IRQ_STATUS);
  R0 = 1 ;
  W[P0] = R0 ;  /* Clear DMA interrupt */

  /* Wait for DMA to complete */
  P0.L = lo(DMA7_IRQ_STATUS);
  P0.H = hi(DMA7_IRQ_STATUS);
  R0 = DMA_RUN;      /* 0x08 */

CHECK_DMA_COMPLETE:  /* Poll for DMA_RUN bit to clear */
  R3 = W[P0] (Z);
  R1 = R3 & R0;
  CC = R1 == 0;
  IF !CC JUMP CHECK_DMA_COMPLETE;

  /* Wait for TXS to clear */
  P0.L = lo(SPI_STAT);
  P0.H = hi(SPI_STAT);
  R1 = TXS;   /* 0x08 */

Check_TXS:    /* Poll for TXS = 0 */
  R2 = W[P0] (Z);
  R2 = R2 & R1;
  CC = R0 == 0;
  IF !CC JUMP Check_TXS;

  R2 = W[P0] (Z);  /* Check if TXS stays clear for 2 reads */
  R2 = R2 & R1;
  CC = R0 == 0;
```

```
  IF !CC JUMP Check_TXS;

/* Wait for final word to transmit from SPI */
Final_Word:
  R0 = W[P0](Z);
  R2 = SPIF;   /* 0x01 */
  R0 = R0 & R2;
  CC = R0 == 0;
  IF CC JUMP Final_Word;

Disable_SPI:
  P0.L = lo(SPI_CTL);
  P0.H = hi(SPI_CTL);
  R0 = W[P0] (Z);
  BITCLR (R0,0xe);   /* Clear SPI enable bit */
  W[P0] = R0;        /* Disable SPI */

Disable_DMA:
  P0.L = lo(DMA7_CONFIG);
  P0.H = hi(DMA7_CONFIG);
  R0 = W[P0](Z);
  BITCLR (R0,0x0);   /* Clear DMA enable bit */
  W[P0] = R0;        /* Disable DMA */

  RTI;   /* Exit Handler */
```

Unique Behavior for the ADSP-BF52x Processor

None.

23 TWO WIRE INTERFACE CONTROLLER

This chapter describes the two wire interface (TWI) port. Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF52x

For details regarding the number of TWIs for the ADSP-BF52x product, please refer to the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet*.

For TWI interrupt vector assignments, refer to Table 5-3 on page 5-18 in Chapter 5, "System Interrupts".

To determine how each of the TWIs is multiplexed with other functional pins, refer to Table 9-2 on page 9-5 through Table 9-5 on page 9-9 in Chapter 9, "General-Purpose Ports".

For a list of MMR addresses for each TWI, refer to Appendix A, "System MMR Assignments".

TWI behavior for the ADSP-BF52x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF52x Processor" on page 23-60.

Overview

The TWI controller allows a device to interface to an inter IC bus as specified by the *Philips I²C Bus Specification version 2.1* dated January 2000.

The TWI is fully compatible with the widely used I²C bus standard. It was designed with a high level of functionality and is compatible with multi-master, multi-slave bus configurations. To preserve processor bandwidth the TWI controller can be set up with transfer initiated interrupts only to service FIFO buffer data reads and writes. Protocol related interrupts are optional.

The TWI externally moves 8-bit data while maintaining compliance with the I²C bus protocol. The TWI controller includes these features:

- Simultaneous master and slave operation on multiple device systems
- Support for multi-master bus arbitration
- 7-bit addressing
- 100K bits/second and 400K bits/second data rates
- General call address support
- Master clock synchronization and support for clock low extension
- Separate multiple-byte receive and transmit FIFOs
- Low interrupt rate
- Individual override control of data and clock lines in the event of bus lock-up

- Input filter for spike suppression
- Serial camera control bus support as specified in the *OmniVision Serial Camera Control Bus (SCCB) Functional Specification* version 2.1.

Interface Overview

Figure 23-1 provides a block diagram of the TWI controller. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the SCL rate, to and from other TWI devices. The SCL signal synchronizes the shifting and sampling of the data on the serial data pin.

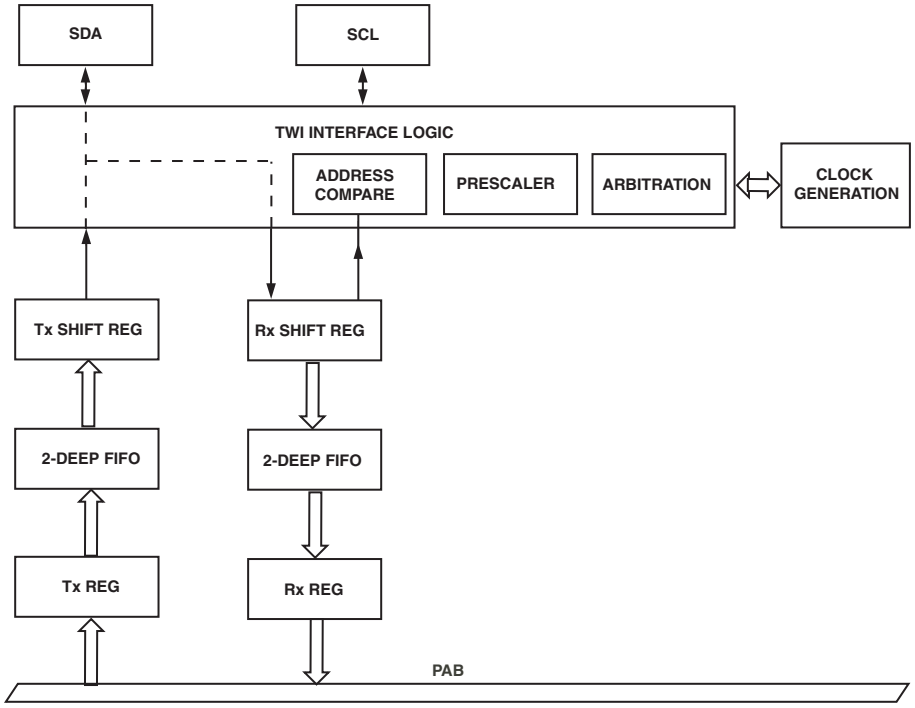


Figure 23-1. TWI Block Diagram

External Interface

The SDA (serial data) and SCL (serial clock) signals are open drain and as such require pull-up resistors.

Serial Clock Signal (SCL)

In slave mode this signal is an input and an external master is responsible for providing the clock.

In master mode the TWI controller must set this signal to the desired frequency. The TWI controller supports the standard mode of operation (up to 100 KHz) or fast mode (up to 400 KHz).

The TWI control register (TWI_CONTROL) is used to set the PRESCALE value which gives the relationship between the system clock (SCLK) and the TWI controller's internally timed events. The internal time reference is derived from SCLK using a prescaled value.

$$\text{PRESCALE} = f_{SCLK}/10\text{MHz}$$

The PRESCALE value is the number of system clock (SCLK) periods used in the generation of one internal time reference. The value of PRESCALE must be set to create an internal time reference with a period of 10 MHz. It is represented as a 7-bit binary value.

Serial Data Signal (SDA)

This is a bidirectional signal on which serial data is transmitted or received depending on the direction of the transfer.

TWI Pins

Table 23-1 shows the pins for the TWI. Two bidirectional pins externally interface the TWI controller to the I^2C bus. The interface is simple and no other external connections or logic are required.

Table 23-1. TWI Pins

Pin	Description
SDA	In/Out TWI serial data, high impedance reset value.
SCL	In/Out TWI serial clock, high impedance reset value.

Internal Interfaces

The peripheral bus interface supports the transfer of 16-bit wide data and is used by the processor in the support of register and FIFO buffer reads and writes.

The register block contains all control and status bits and reflects what can be written or read as outlined by the programmer's model. Status bits can be updated by their respective functional blocks.

The FIFO buffer is configured as a1-byte-wide 2-deep transmit FIFO buffer and a 1-byte-wide 2-deep receive FIFO buffer.

The transmit shift register serially shifts its data out externally off chip. The output can be controlled for generation of acknowledgements or it can be manually overwritten.

The receive shift register receives its data serially from off chip. The receive shift register is 1 byte wide and data received can either be transferred to the FIFO buffer or used in an address comparison.

The address compare block supports address comparison in the event the TWI controller module is accessed as a slave.

The prescaler block must be programmed to generate a 10 MHz time reference relative to the system clock. This time base is used for filtering of data and timing events specified by the electrical data sheet (See the Philips Specification), as well as for SCL clock generation.

The clock generation module is used to generate an external SCL clock when in master mode. It includes the logic necessary for synchronization in a multi-master clock configuration and clock stretching when configured in slave mode.

Description of Operation

The following sections describe the operation of the TWI interface.

TWI Transfer Protocols

The TWI controller follows the transfer protocol of the *Philips I^2C Bus Specification version 2.1* dated January 2000. A simple complete transfer is diagrammed in Figure 23-2.

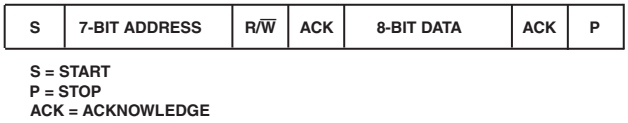


Figure 23-2. Basic Data Transfer

To better understand the mapping of TWI controller register contents to a basic transfer, Figure 23-3 details the same transfer as above noting the corresponding TWI controller bit names. In this illustration, the TWI

controller successfully transmits one byte of data. The slave has acknowledged both address and data.

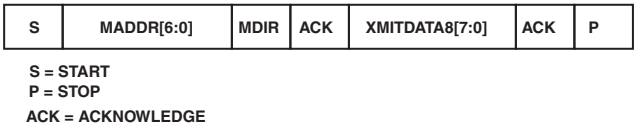


Figure 23-3. Data Transfer With Bit Illustration

Clock Generation and Synchronization

The TWI controller implementation only issues a clock during master mode operation and only at the time a transfer has been initiated. If arbitration for the bus is lost, the serial clock output immediately three-states. If multiple clocks attempt to drive the serial clock line, the TWI controller synchronizes its clock with the other remaining clocks. This is shown in Figure 23-4.

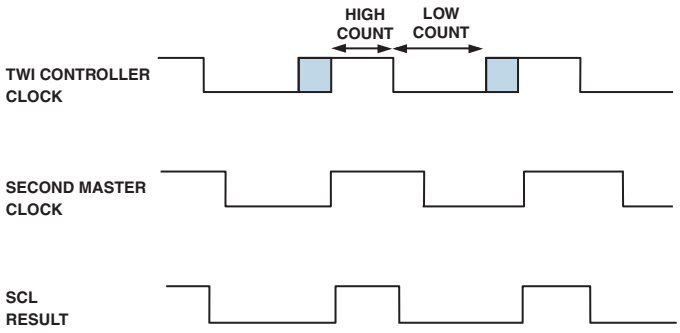


Figure 23-4. TWI Clock Synchronization

The TWI controller's serial clock (SCL) output follows these rules:

- Once the clock high (CLKHI) count is complete, the serial clock output is driven low and the clock low (CLKLOW) count begins.
- Once the clock low count is complete, the serial clock line is three-stated and the clock synchronization logic enters into a delay mode (shaded area) until the SCL line is detected at a logic 1 level. At this time the clock high count begins.

Bus Arbitration

The TWI controller initiates a master mode transmission (MEN) only when the bus is idle. If the bus is idle and two masters initiate a transfer, arbitration for the bus begins. This is shown in Figure 23-5.

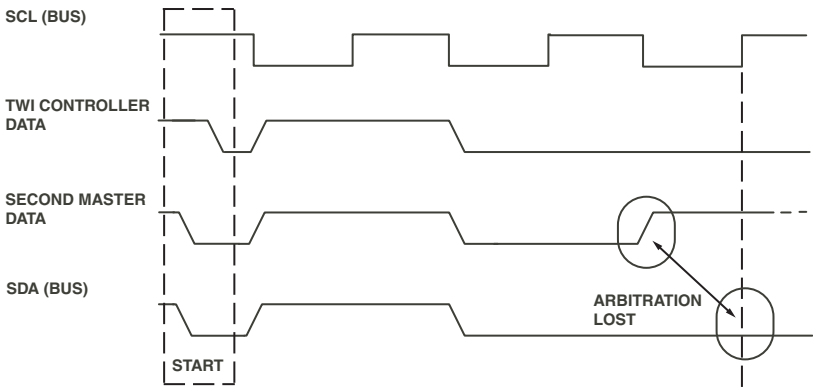


Figure 23-5. TWI Bus Arbitration

The TWI controller monitors the serial data bus (SDA) while SCL is high and if SDA is determined to be an active logic 0 level while the TWI controller's data is a logic 1 level, the TWI controller has lost arbitration and ends generation of clock and data. Note arbitration is not performed only at serial clock edges, but also during the entire time SCL is high.

Start and Stop Conditions

Start and stop conditions involve serial data transitions while the serial clock is a logic 1 level. The TWI controller generates and recognizes these transitions. Typically start and stop conditions occur at the beginning and at the conclusion of a transmission with the exception repeated start "combined" transfers, as shown in Figure 23-6.

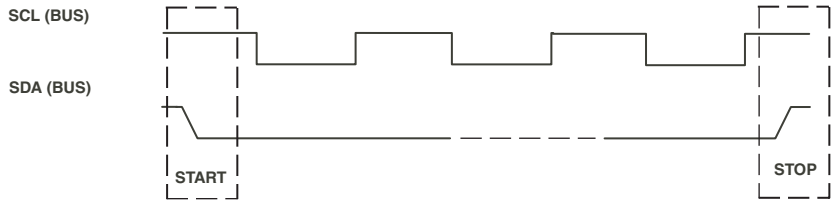


Figure 23-6. TWI Start and Stop Conditions

The TWI controller's special case start and stop conditions include:

- TWI controller addressed as a slave-receiver

  If the master asserts a stop condition during the data phase of a transfer, the TWI controller concludes the transfer (SCOMP).

- TWI controller addressed as a slave-transmitter

  If the master asserts a stop condition during the data phase of a transfer, the TWI controller concludes the transfer (SCOMP) and indicates a slave transfer error (SERR).

- TWI controller as a master-transmitter or master-receiver

  If the stop bit is set during an active master transfer, the TWI controller issues a stop condition as soon as possible avoiding any error conditions (as if data transfer count had been reached).

General Call Support

The TWI controller always decodes and acknowledges a general call address if it is enabled as a slave (SEN) and if general call is enabled (GEN). general call addressing (0x00) is indicated by the GCALL bit being set and by nature of the transfer the TWI controller is a slave-receiver. If the data associated with the transfer is to be NAK'ed, the NAK bit can be set.

If the TWI controller is to issue a general call as a master-transmitter the appropriate address and transfer direction can be set along with loading transmit FIFO data.

Fast Mode

Fast mode essentially uses the same mechanics as standard mode of operation. It is the electrical specifications and timing that are most affected. When fast mode is enabled (FAST) the following timings are modified to meet the electrical requirements.

- Serial data rise times before arbitration evaluation (t_r)
- Stop condition set-up time from serial clock to serial data ($t_{SU;STO}$)
- Bus free time between a stop and start condition (t_{BUF})

Functional Description

The following sections describe the functional operation of the TWI.

General Setup

General setup refers to register writes that are required for both slave mode operation and master mode operation. General setup should be performed before either the master or slave enable bits are set.

- Program the TWI_CONTROL register to enable the TWI controller and set the prescale value. Program the prescale value to the binary representation of f_{SCLK}/10MHz

All values should be rounded up to the next whole number. The TWI_ENA bit enable must be set. Note once the TWI controller is enabled a bus busy condition may be detected. This condition should clear after t_{BUF} has expired assuming no additional bus activity has been detected.

Slave Mode

When enabled, slave mode operation supports both receive and transmit data transfers. It is not possible to enable only one data transfer direction and not acknowledge (NAK) the other. This is reflected in the following setup.

1. Program TWI_SLAVE_ADDR. The appropriate 7 bits are used in determining a match during the address phase of the transfer.
2. Program TWI_XMT_DATA8 or TWI_XMT_DATA16. These are the initial data values to be transmitted in the event the slave is addressed and a transmit is required. This is an optional step. If no data is written and the slave is addressed and a transmit is required, the serial clock (SCL) is stretched and an interrupt is generated until data is written to the transmit FIFO.
3. Program TWI_INT_MASK. Enable bits are associated with the desired interrupt sources. As an example, programming the value 0x000F results in an interrupt output to the processor in the event that a

valid address match is detected, a valid slave transfer completes, a slave transfer has an error, a subsequent transfer has begun yet the previous transfer has not been serviced.

4. Program `TWI_SLAVE_CTL`. Ultimately this prepares and enables slave mode operation. As an example, programming the value 0x0005 enables slave mode operation, requires 7-bit addressing, and indicates that data in the transmit FIFO buffer is intended for slave mode transmission.

Table 23-2 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 23-2. Slave Mode Setup Interaction

TWI Controller Master	Processor
Interrupt: SINIT – Slave transfer in progress.	Acknowledge: Clear interrupt source bits.
Interrupt: RCVFULL – Receive buffer is full.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
Interrupt: SCOMP – Slave transfer complete.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.

Master Mode Clock Setup

Master mode operation is set up and executed on a per-transfer basis. An example of programming steps for a receive and for a transmit are given separately in following sections. The clock setup programming step listed here is common to both transfer types.

- Program `TWI_CLKDIV`. This defines the clock high duration and clock low duration.

Master Mode Transmit

Follow these programming steps for a single master mode transmit:

1. Program TWI_MASTER_ADDR. This defines the address transmitted during the address phase of the transfer.
2. Program TWI_XMT_DATA8 or TWI_XMT_DATA16. This is the initial data transmitted. It is considered an error to complete the address phase of the transfer and not have data available in the transmit FIFO buffer.
3. Program TWI_FIFO_CTL. Indicate if transmit FIFO buffer interrupts should occur with each byte transmitted (8-bits) or with each two bytes transmitted (16-bits).
4. Program TWI_INT_MASK. Enable bits associated with the desired interrupt sources. As an example, programming the value 0x0030 results in an interrupt output to the processor in the event that the master transfer completes, and the master transfer has an error.
5. Program TWI_MASTER_CTL. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0201 enables master mode operation, generates a 7-bit address, sets the direction to master-transmit, uses standard mode timing, and transmits 8 data bytes before generating a Stop condition.

Table 23-3 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 23-3. Master Mode Transmit Setup Interaction

TWI Controller Master	Processor
Interrupt: XMTEMPTY – Transmit buffer is empty.	Write transmit FIFO buffer. Acknowledge: Clear interrupt source bits.

Table 23-3. Master Mode Transmit Setup Interaction (Continued)

TWI Controller Master	Processor
...	...
Interrupt: MCOMP – Master transfer complete.	Acknowledge: Clear interrupt source bits.

Master Mode Receive

Follow these programming steps for a single master mode receive:

1. Program `TWI_MASTER_ADDR`. This defines the address transmitted during the address phase of the transfer.
2. Program `TWI_FIFO_CTL`. Indicate if receive FIFO buffer interrupts should occur with each byte received (8-bits) or with each two bytes received (16-bits).
3. Program `TWI_INT_MASK`. Enable bits associated with the desired interrupt sources. For example, programming the value 0x0030 results in an interrupt output to the processor in the event that the master transfer completes, and the master transfer has an error.
4. Program `TWI_MASTER_CTL`. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0205 enables master mode operation, generates a 7-bit address, sets the direction to master-receive, uses standard mode timing, and receives 8 data bytes before generating a Stop condition.

Table 23-4 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 23-4. Master Mode Receive Setup Interaction

TWI Controller Master	Processor
Interrupt: RCVFULL – Receive buffer is full.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
Interrupt: MCOMP – Master transfer complete.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.

Repeated Start Condition

In general, a repeated start condition is the absence of a stop condition between two transfers. The two transfers can be of any direction type. Examples include a transmit followed by a receive, or a receive followed by a transmit. The following sections guide the programmer in developing a service routine.

Transmit/Receive Repeated Start Sequence

Figure 23-7 shows a repeated start data transmit followed by a data receive sequence.

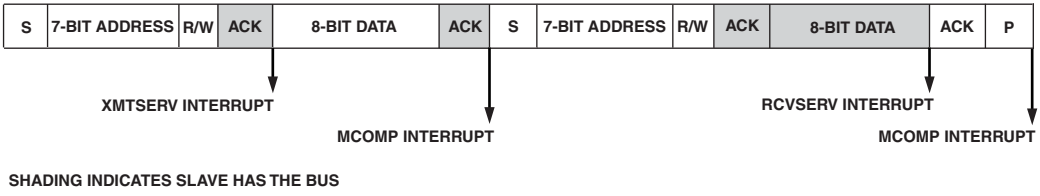


Figure 23-7. Transmit/Receive Data Repeated Start

The following tasks are performed at each interrupt.

- `XMTSERV` interrupt

  This interrupt was generated due to a FIFO access. Since this is the last byte of this transfer, `FIFO_STATUS` indicates the transmit FIFO is empty. When read, `DCNT` would be zero. Set the `RSTART` bit to indicate a repeated start and set the `MDIR` bit if the following transfer will be a data receive.

- `MCOMP` interrupt

  This interrupt was generated because all data has been transferred (`DCNT` = 0). If no errors were generated, a start condition is initiated. Clear the `RSTART` bit and program the `DCNT` with the desired number of bytes to receive.

- `RCVSERV` interrupt

  This interrupt is generated due to the arrival of a byte in the receive FIFO. Simple data handling is all that is required.

- `MCOMP` interrupt

  The transfer is complete.

Receive/Transmit Repeated Start Sequence

Figure 23-8 on page 23-17 illustrates a repeated start data receive followed by a data transmit sequence.

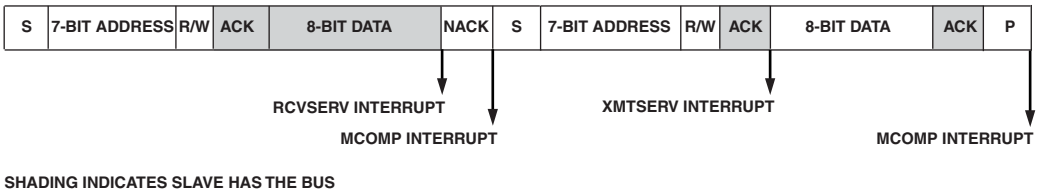


Figure 23-8. Receive/Transmit Data Repeated Start

The tasks performed at each interrupt are:

- `RCVSERV` interrupt

  This interrupt is generated due to the arrival of a data byte in the receive FIFO. Set the `RSTART` bit to indicate a repeated start and clear the `MDIR` bit if the following transfer will be a data transmit.

- `MCOMP` interrupt

  This interrupt has occurred due to the completion of the data receive transfer. If no errors were generated, a start condition is initiated. Clear the `RSTART` bit and program the `DCNT` with the desired number of bytes to transmit.

- `XMTSERV` interrupt

  This interrupt is generated due to a FIFO access. Simple data handling is all that is required.

- `MCOMP` interrupt

The transfer is complete.

There is no timing constraint to meet the above conditions—the user can program the bits as required. Refer to "Clock Stretching During Repeated Start Condition" on page 23-21 for more on how the controller stretches the clock during Repeated Start transfers.

Clock Stretching

Clock stretching is an added functionality of the TWI controller in Master Mode operation. This new behavior utilizes self-induced stretching of the I^2C clock while waiting on servicing interrupts. Stretching is done automatically by the hardware and no programming is required for this.

The TWI Controller as Master supports three modes of clock stretching: "Clock Stretching During FIFO Underflow" on page 23-18, "Clock Stretching During FIFO Overflow" on page 23-20 and "Clock Stretching During Repeated Start Condition" on page 23-21.

Clock Stretching During FIFO Underflow

During a master mode transmit, an interrupt is generated at the instant the transmit FIFO becomes empty. At this time, the most recent byte begins transmission. If the XMTSERV interrupt is not serviced, the concluding "acknowledge" phase of the transfer will be stretched. Stretching of the clock continues until new data bytes are written to the transmit FIFO (`TWI_XMT_DATA8` or `TWI_XMT_DATA16`). No other action is required to release the clock and continue the transmission. This behavior continues until the transmission is complete (`DCNT` = 0) at which time the transmis-

sion is concluded (MCOMP) as shown in Figure 23-9 and described in Table 23-5.

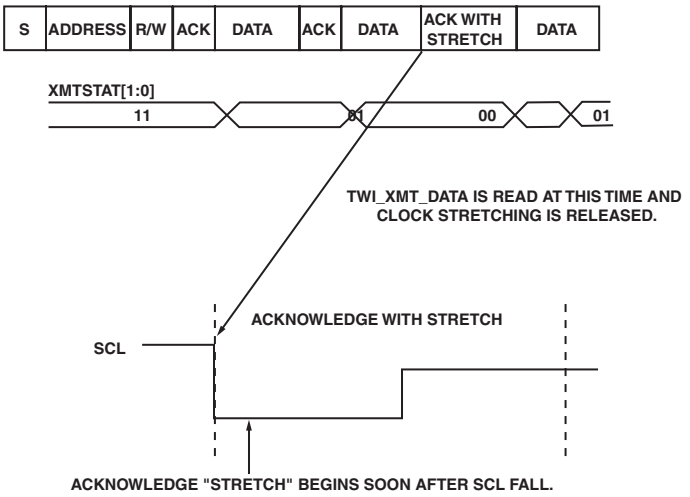


Figure 23-9. Clock Stretching during FIFO Underflow

Table 23-5. FIFO Underflow Case

TWI Controller	Processor
Interrupt: XMTSERV – Transmit FIFO buffer is empty.	Acknowledge: Clear interrupt source bits. Write transmit FIFO buffer.
...	...
Interrupt: MCOMP – Master transmit complete (DCNT= 0x00).	Acknowledge: Clear interrupt source bits.

Clock Stretching During FIFO Overflow

During a master mode receive, an interrupt is generated at the instant the receive FIFO becomes full. It is during the acknowledge phase of this received byte that clock stretching begins. No attempt is made to initiate the reception of an additional byte. Stretching of the clock continues until the data bytes previously received are read from the receive FIFO buffer (`TWI_RCV_DATA8`, `TWI_RCV_DATA16`). No other action is required to release the clock and continue the reception of data. This behavior continues until the reception is complete (`DCNT` = 0x00) at which time the reception is concluded (`MCOMP`) as shown in Figure 23-10 and described in Table 23-6.

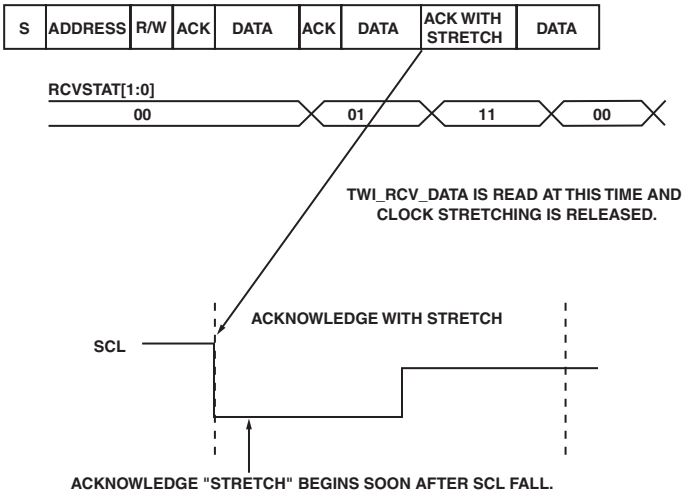


Figure 23-10. Clock Stretching During FIFO Overflow

Table 23-6. FIFO Overflow Case

TWI Controller	Processor
Interrupt: RCVSERV – Receive FIFO buffer is full.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.
...	...
Interrupt: MCOMP – Master receive complete.	Acknowledge: Clear interrupt source bits.

Clock Stretching During Repeated Start Condition

The repeated start feature in I^2C protocol requires transitioning between two subsequent transfers. With the use of clock stretching, the task of managing transitions becomes simpler and becomes common to all transfer types.

Once an initial TWI master transfer has completed (transmit or receive) the clock will initiate a stretch during the repeated start phase between

transfers. Concurrent with this event the initial transfer will generate a transfer complete interrupt (MCOMP) to signify the initial transfer has completed (DCNT = 0). This initial transfer is handled without any special bit setting sequences or timings. The clock stretching logic described above applies here. With no system related timing constraints the subsequent transfer (receive or transmit) is setup and activated. This sequence can be repeated as many times as required to string a series of repeated start transfers together. This is shown in Figure 23-11 and described in Table 23-7.

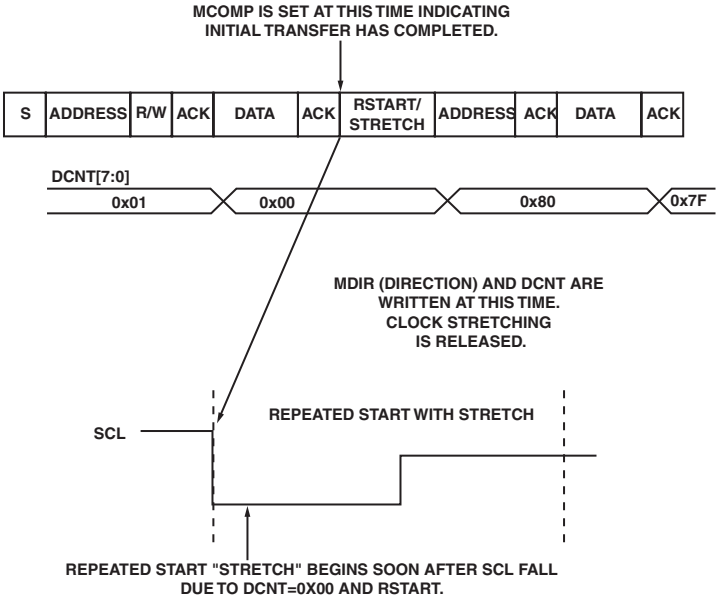


Figure 23-11. Clock Stretching during Repeated Start Condition

Table 23-7. Repeated Start Case

TWI Controller	Processor
Interrupt: MCOMP – Initial transmit has completed and DCNT = 0x00. Note: transfer in progress, RSTART previously set.	Acknowledge: Clear interrupt source bits. Write TWI_MASTER_CTL, setting MDIR (receive), clearing RSTART, and setting new DCNT value (nonzero).
Interrupt: RCVSERV – Receive FIFO is full.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.
...	...
Interrupt: MCOMP – Master receive complete.	Acknowledge: Clear interrupt source bits.

Programming Model

Figure 23-12 and Figure 23-13 illustrate the programming model for the TWI.

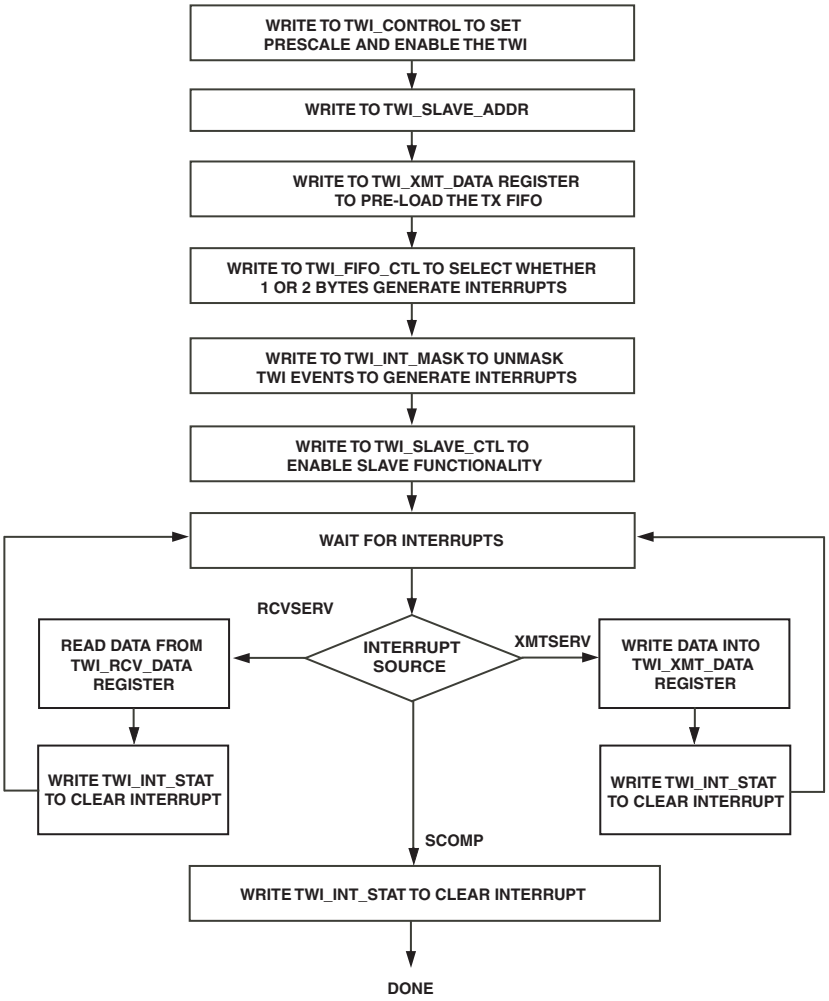

WRITE TO TWI_CONTROL TO SET PRESCALE AND ENABLE THE TWI
WRITE TO TWI_SLAVE_ADDR
WRITE TO TWI_XMT_DATA REGISTER TO PRE-LOAD THE TX FIFO
WRITE TO TWI_FIFO_CTL TO SELECT WHETHER 1 OR 2 BYTES GENERATE INTERRUPTS
WRITE TO TWI_INT_MASK TO UNMASK TWI EVENTS TO GENERATE INTERRUPTS
WRITE TO TWI_SLAVE_CTL TO ENABLE SLAVE FUNCTIONALITY
WAIT FOR INTERRUPTS
RCVSERV
READ DATA FROM TWI_RCV_DATA REGISTER
INTERRUPT SOURCE
XMTSERV
WRITE DATA INTO TWI_XMT_DATA REGISTER
WRITE TWI_INT_STAT TO CLEAR INTERRUPT
WRITE TWI_INT_STAT TO CLEAR INTERRUPT
SCOMP
WRITE TWI_INT_STAT TO CLEAR INTERRUPT
DONE

Figure 23-12. TWI Slave Mode

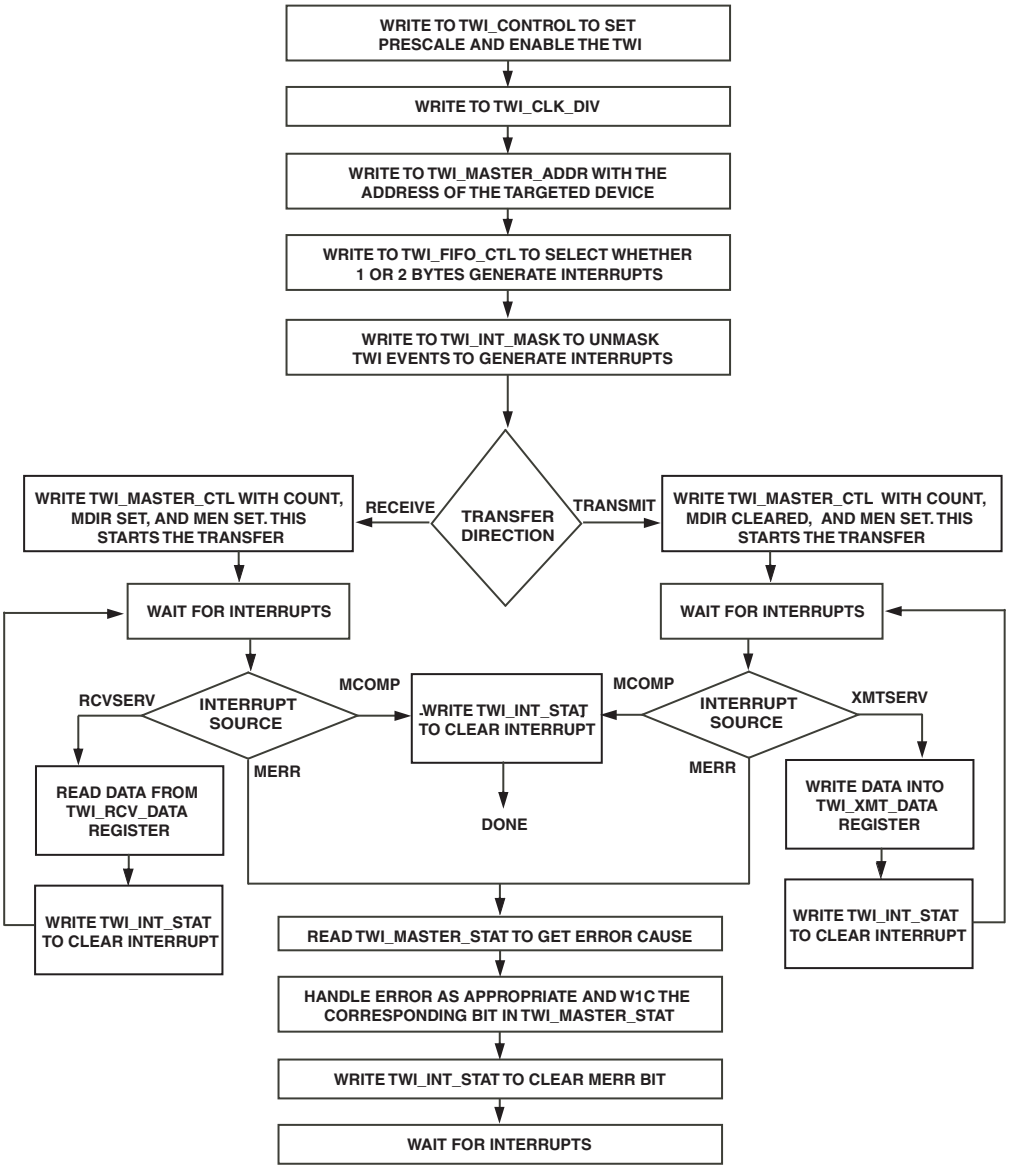

WRITE TO TWI_CONTROL TO SET PRESCALE AND ENABLE THE TWI
WRITE TO TWI_CLK_DIV
WRITE TO TWI_MASTER_ADDR WITH THE ADDRESS OF THE TARGETED DEVICE
WRITE TO TWI_FIFO_CTL TO SELECT WHETHER 1 OR 2 BYTES GENERATE INTERRUPTS
WRITE TO TWI_INT_MASK TO UNMASK TWI EVENTS TO GENERATE INTERRUPTS
TRANSFER DIRECTION
RECEIVE
TRANSMIT
WRITE TWI_MASTER_CTL WITH COUNT, MDIR SET, AND MEN SET. THIS STARTS THE TRANSFER
WRITE TWI_MASTER_CTL WITH COUNT, MDIR CLEARED, AND MEN SET. THIS STARTS THE TRANSFER
WAIT FOR INTERRUPTS
WAIT FOR INTERRUPTS
INTERRUPT SOURCE
RCVSERV
MCOMP
MERR
WRITE TWI_INT_STAT TO CLEAR INTERRUPT
MCOMP
INTERRUPT SOURCE
XMTSERV
MERR
READ DATA FROM TWI_RCV_DATA REGISTER
DONE
WRITE DATA INTO TWI_XMT_DATA REGISTER
WRITE TWI_INT_STAT TO CLEAR INTERRUPT
WRITE TWI_INT_STAT TO CLEAR INTERRUPT
READ TWI_MASTER_STAT TO GET ERROR CAUSE
HANDLE ERROR AS APPROPRIATE AND W1C THE CORRESPONDING BIT IN TWI_MASTER_STAT
WRITE TWI_INT_STAT TO CLEAR MERR BIT
WAIT FOR INTERRUPTS

Figure 23-13. TWI Master Mode

Register Descriptions

The TWI controller has 16 registers described in the following sections. Figure 23-14 through Figure 23-31 on page 23-48 illustrate the registers.

TWI CONTROL Register (TWI_CONTROL)

The TWI_CONTROL register is used to enable the TWI module as well as to establish a relationship between the system clock (SCLK) and the TWI controller's internally timed events. The internal time reference is derived from SCLK using a prescaled value.

PRESCALE = f_{SCLK}/10MHz

SCCB compatibility is an optional feature and should not be used in an I^2C bus system. This feature is turned on by setting the SCCB bit in the TWI_CONTROL register. When this feature is set all slave asserted acknowledgement bits are ignored by this master. This feature is valid only during transfers where the TWI is mastering an SCCB bus. Slave mode transfers should be avoided when this feature is enabled because the TWI controller always generates an acknowledge in slave mode.

For either master and/or slave mode of operation, the TWI controller is enabled by setting the TWI_ENA bit in the TWI_CONTROL register. It is recommended that this bit be set at the time PRESCALE is initialized and remain set. This guarantees accurate operation of bus busy detection logic.

The PRESCALE field of the TWI_CONTROL register specifies the number of system clock (SCLK) periods used in the generation of one internal time

reference. The value of PRESCALE must be set to create an internal time reference with a period of 10 MHz. It is represented as a 7-bit binary value.

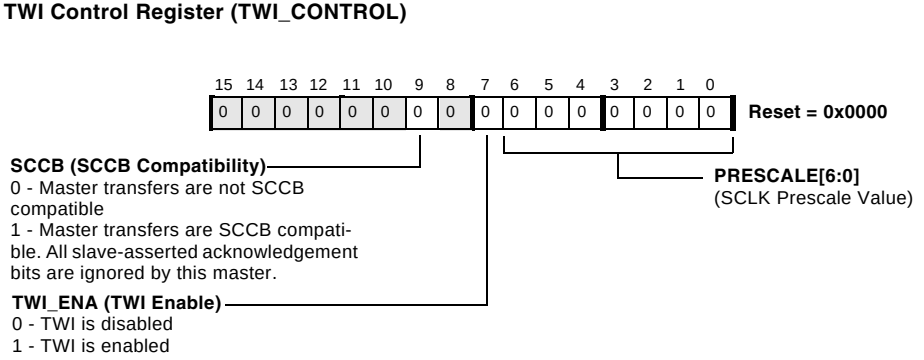

TWI Control Register (TWI_CONTROL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SCCB (SCCB Compatibility)
0 - Master transfers are not SCCB compatible
1 - Master transfers are SCCB compatible. All slave-asserted acknowledgement bits are ignored by this master.
TWI_ENA (TWI Enable)
0 - TWI is disabled
1 - TWI is enabled
PRESCALE[6:0]
(SCLK Prescale Value)

Figure 23-14. TWI Control Register

SCL Clock Divider Register (TWI_CLKDIV)

The clock signal SCL is an output in master mode and an input in slave mode.

During master mode operation, the TWI_CLKDIV register values are used to create the high and low durations of the serial clock (SCL). Serial clock frequencies can vary from 400 KHz to less than 20 KHz. The resolution of the clock generated is 1/10 MHz or 100 ns.

CLKDIV = TWI SCL period / 10 MHz time reference

For example, for an SCL of 400 KHz (period = 1/400 KHz = 2500 ns) and an internal time reference of 10 MHz (period = 100 ns):

CLKDIV = 2500 ns / 100 ns = 25

For an SCL with a 30% duty cycle, then CLKLOW = 17 and CLKHI = 8. Note that CLKLOW and CLKHI add up to CLKDIV.

The CLKHI field of the TWI_CLKDIV register specifies the number of 10 MHz time reference periods the serial clock (SCL) waits before a new clock low period begins, assuming a single master. It is represented as an 8-bit binary value.

The CLKLOW field of the TWI_CLKDIV register specifies the number of internal time reference periods the serial clock (SCL) is held low. It is represented as an 8-bit binary value.

SCL Clock Divider Register (TWI_CLKDIV)

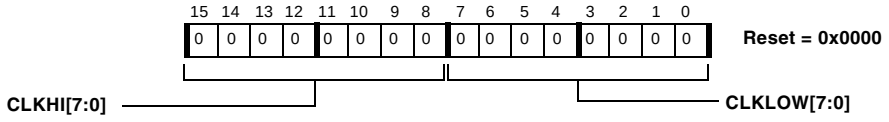


Figure 23-15. SCL Clock Divider Register

TWI Slave Mode Control Register (TWI_SLAVE_CTL)

The TWI_SLAVE_CTL register controls the logic associated with slave mode operation. Settings in this register do not affect master mode operation and should not be modified to control master mode functionality.

TWI Slave Mode Control Register (TWI_SLAVE_CTL)

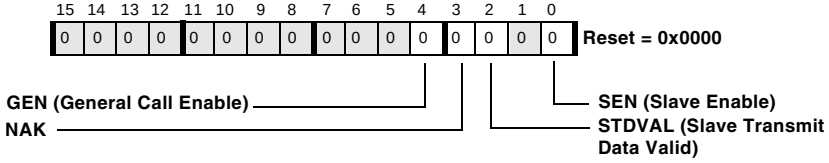

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
GEN (General Call Enable)
NAK
SEN (Slave Enable)
STDVAL (Slave Transmit Data Valid)

Figure 23-16. TWI Slave Mode Control Register

Additional information for the TWI_SLAVE_CTL register bits includes:

- **General call enable** (GEN)

  General call address detection is available only when slave mode is enabled.

  [0] General call address matching is not enabled.

  [1] General call address matching is enabled. A general call slave receive transfer is accepted. All status and interrupt source bits associated with transfers are updated.

- **NAK** (NAK)

  [0] Slave receive transfers generate an ACK at the conclusion of a data transfer.

  [1] Slave receive transfers generate a data NAK (not acknowledge) at the conclusion of a data transfer. The slave is still considered to be addressed.

- **Slave transmit data valid** (STDVAL)

  [0] Data in the transmit FIFO is for master mode transmits and is not allowed to be used during a slave transmit, and the transmit FIFO is treated as if it is empty.

  [1] Data in the transmit FIFO is available for a slave transmission.

- **Slave enable** (SEN)

  [0] The slave is not enabled. No attempt is made to identify a valid address. If cleared during a valid transfer, clock stretching ceases, the serial data line is released, and the current byte is not acknowledged.

  [1] The slave is enabled. Enabling slave and master modes of operation concurrently is allowed.

TWI Slave Mode Address Register (TWI_SLAVE_ADDR)

The TWI_SLAVE_ADDR register holds the slave mode address, which is the valid address that the slave-enabled TWI controller responds to. The TWI controller compares this value with the received address during the addressing phase of a transfer.

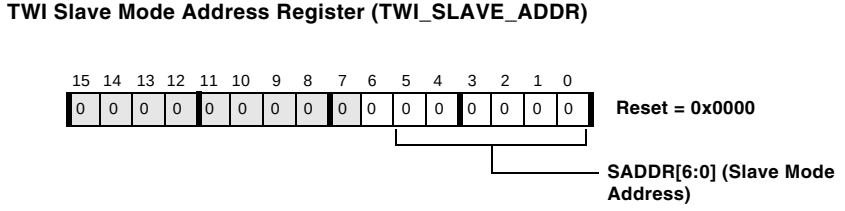

TWI Slave Mode Address Register (TWI_SLAVE_ADDR)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SADDR[6:0] (Slave Mode Address)

Figure 23-17. TWI Slave Mode Address Register

TWI Slave Mode Status Register (TWI_SLAVE_STAT)

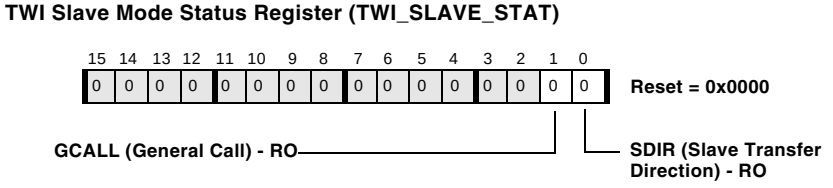

TWI Slave Mode Status Register (TWI_SLAVE_STAT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
GCALL (General Call) - RO
SDIR (Slave Transfer Direction) - RO

Figure 23-18. TWI Slave Mode Status Register

During and at the conclusion of register slave mode transfers, the TWI_SLAVE_STAT register holds information on the current transfer. Gener-

ally slave mode status bits are not associated with the generation of interrupts. Master mode operation does not affect slave mode status bits.

- **General call** (GCALL)

  This bit self clears if slave mode is disabled (SEN = 0).

  [0] At the time of addressing, the address was not determined to be a general call.

  [1] At the time of addressing, the address was determined to be a general call.

- **Slave transfer direction** (SDIR)

  This bit self clears if slave mode is disabled (SEN = 0).

  [0] At the time of addressing, the transfer direction was determined to be slave receive.

  [1] At the time of addressing, the transfer direction was determined to be slave transmit.

TWI Master Mode Control Register (TWI_MASTER_CTL)

The TWI_MASTER_CTL register controls the logic associated with master mode operation. Bits in this register do not affect slave mode operation and should not be modified to control slave mode functionality.

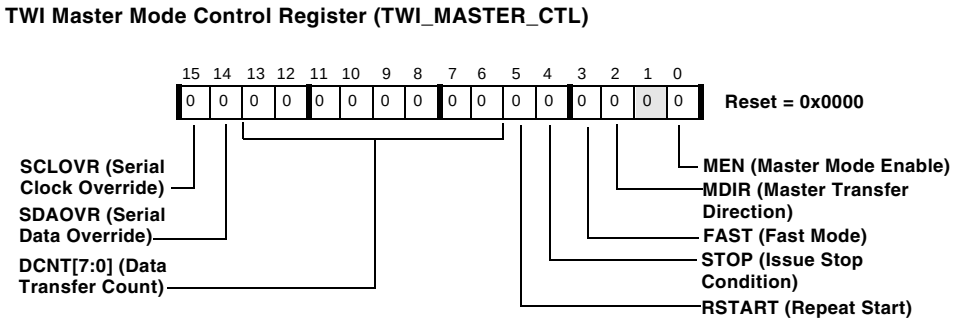

TWI Master Mode Control Register (TWI_MASTER_CTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SCLOVR (Serial Clock Override)
SDAOVR (Serial Data Override)
DCNT[7:0] (Data Transfer Count)
MEN (Master Mode Enable)
MDIR (Master Transfer Direction)
FAST (Fast Mode)
STOP (Issue Stop Condition)
RSTART (Repeat Start)

Figure 23-19. TWI Master Mode Control Register

Additional information for the TWI_MASTER_CTL register bits includes:

- **Serial clock override** (SCLOVR)

  This bit can be used when direct control of the serial clock line is required. Normal master and slave mode operation should not require override operation.

  [0] Normal serial clock operation under the control of master mode clock generation and slave mode clock stretching logic.

  [1] Serial clock output is driven to an active 0 level overriding all other logic. This state is held until this bit is cleared.

- **Serial data (SDA) override** (SDAOVR)

This bit can be used when direct control of the serial data line is required. Normal master and slave mode operation should not require override operation.

[0] Normal serial data operation under the control of the transmit shift register and acknowledge logic.

[1] Serial data output is driven to an active 0 level overriding all other logic. This state is held until this bit is cleared.

- **Data transfer count** (DCNT[7:0])

  Indicates the number of data bytes to transfer. As each data word is transferred, DCNT is decremented. When DCNT is 0, a stop condition is generated. Setting DCNT to 0xFF disables the counter. In this transfer mode, data continues to be transferred until it is concluded by setting the STOP bit.

- **Repeat start** (RSTART)

  [0] Transfer concludes with a stop condition.

  [1] Issue a repeat start condition at the conclusion of the current transfer (DCNT = 0) and begin the next transfer. The current transfer concludes with updates to the appropriate status and interrupt bits. If errors occurred during the previous transfer, a repeat start does not occur. In the absence of any errors, master enable (MEN) does not self clear on a repeat start.

- **Issue stop condition** (STOP)

  [0] Normal transfer operation.

  [1] The transfer concludes as soon as possible avoiding any error conditions (as if data transfer count had been reached) and at that time the TWI interrupt mask register (TWI_INT_MASK) is updated along with any associated status bits.

- **Fast mode** (FAST)

  [0] Standard mode (up to 100K bits/s) timing specifications in use.

  [1] Fast mode (up to 400K bits/s) timing specifications in use.

- **Master transfer direction** (MDIR)

  [0] The initiated transfer is master transmit.

  [1] The initiated transfer is master receive.

- **Master mode enable** (MEN)

  This bit self clears at the completion of a transfer. This includes transfers terminated due to errors.

  [0] Master mode functionality is disabled. If this bit is cleared during operation, the transfer is aborted and all logic associated with master mode transfers are reset. Serial data and serial clock (SDA, SCL) are no longer driven. Write-1-to-clear status bits are not affected.

  [1] Master mode functionality is enabled. A start condition is generated if the bus is idle.

TWI Master Mode Address Register (TWI_MASTER_ADDR)

During the addressing phase of a transfer, the TWI controller, with its master enabled, transmits the contents of the TWI_MASTER_ADDR register. When programming this register, omit the read/write bit. That is, only the upper 7 bits that make up the slave address should be written to this register. For example, if the slave address is b#1010000X, where X is the read/write bit, then TWI_MASTER_ADDR is programmed with b#1010000, which corresponds to 0x50. When sending out the address on the bus, the

TWI controller appends the read/write bit as appropriate based on the state of the MDIR bit in the master mode control register.

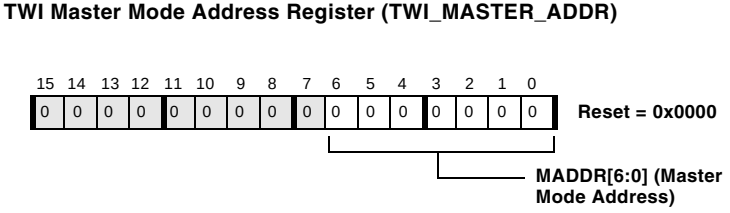

TWI Master Mode Address Register (TWI_MASTER_ADDR)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
MADDR[6:0] (Master Mode Address)

Figure 23-20. TWI Master Mode Address Register

TWI Master Mode Status Register (TWI_MASTER_STAT)

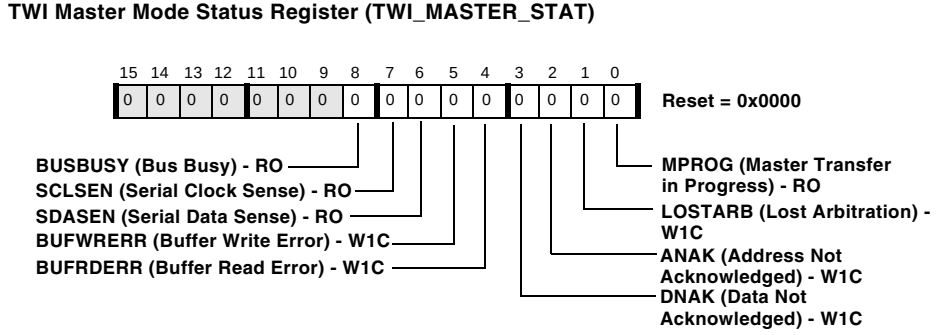

TWI Master Mode Status Register (TWI_MASTER_STAT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
BUSBUSY (Bus Busy) - RO
SCLSEN (Serial Clock Sense) - RO
SDASEN (Serial Data Sense) - RO
BUFWRERR (Buffer Write Error) - W1C
BUFRDERR (Buffer Read Error) - W1C
MPROG (Master Transfer in Progress) - RO
LOSTARB (Lost Arbitration) - W1C
ANAK (Address Not Acknowledged) - W1C
DNAK (Data Not Acknowledged) - W1C

Figure 23-21. TWI Master Mode Status Register

The TWI_MASTER_STAT register holds information during master mode transfers and at their conclusion. Generally, master mode status bits are not directly associated with the generation of interrupts but offer information on the current transfer. Slave mode operation does not affect master mode status bits.

Note that—while the SCLSEN bit is set (this could be due to having no pull-up resistor on SCL or another agent is driving SCL low)—the acknowledge bits (ANAK and DNAK) do not update. This result occurs because the acknowledge conditions are sampled during the high phase of SCL.

- **Bus busy** (BUSBUSY)

  Indicates whether the bus is currently busy or free. This indication is not limited to only this device but is for all devices. Upon a start condition, the setting of the register value is delayed due to the input filtering. Upon a stop condition the clearing of the register value occurs after t_{BUF}.

  [0] The bus is free. The clock and data bus signals have been inactive for the appropriate bus free time.

  [1] The bus is busy. Clock or data activity has been detected.

- **Serial clock sense** (SCLSEN)

  This status bit can be used when direct sensing of the serial clock line is required. The register value is delayed due to the input filter (nominally 50 ns). Normal master and slave mode operation should not require this feature.

  [0] An inactive “one” is currently being sensed on the serial clock.

  [1] An active “zero” is currently being sensed on the serial clock. The source of the active driver is not known and can be internal or external.

- **Serial data sense** (SDASEN)

  This status bit can be used when direct sensing of the serial data line is required. The register value is delayed due to the input filter (nominally 50 ns). Normal master and slave mode operation should not require this feature.

[0] An inactive “one” is currently being sensed on the serial data line.

[1] An active “zero” is currently being sensed on the serial data line. The source of the active driver is not known and can be internal or external.

- **Buffer write error** (BUFWRERR)

  [0] The current master receive has not detected a receive buffer write error.

  [1] The current master transfer was aborted due to a receive buffer write error. The receive buffer and receive shift register were both full at the same time. This bit is W1C.

- **Buffer read error** (BUFRDERR)

  [0] The current master transmit has not detected a buffer read error.

  [1] The current master transfer was aborted due to a transmit buffer read error. At the time data was required by the transmit shift register the buffer was empty. This bit is W1C.

- **Data not acknowledged** (DNAK)

  [0] The current master receive has not detected a NAK during data transmission.

  [1] The current master transfer was aborted due to the detection of a NAK during data transmission. This bit is W1C.

- **Address not acknowledged** (ANAK)

  [0] The current master transmit has not detected NAK during addressing.

[1] The current master transfer was aborted due to the detection of a NAK during the address phase of the transfer. This bit is W1C.

- **Lost arbitration** (LOSTARB)

  [0] The current transfer has not lost arbitration with another master.

  [1] The current transfer was aborted due to the loss of arbitration with another master. This bit is W1C.

- **Master transfer in progress** (MPROG)

  [0] Currently no transfer is taking place. This can occur once a transfer is complete or while an enabled master is waiting for an idle bus.

  [1] A master transfer is in progress.

TWI FIFO Control Register (TWI_FIFO_CTL)

The TWI_FIFO_CTL register control bits affect only the FIFO and are not tied in any way with master or slave mode operation.

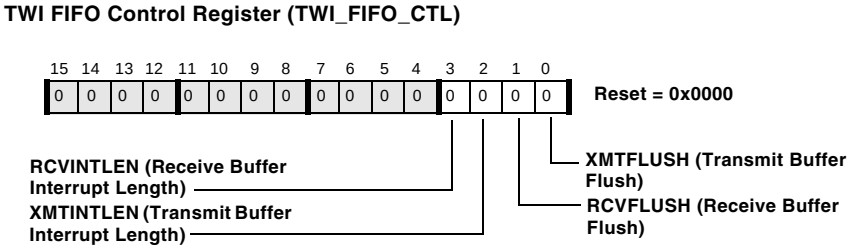

TWI FIFO Control Register (TWI_FIFO_CTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RCVINTLEN (Receive Buffer Interrupt Length)
XMTINTLEN (Transmit Buffer Interrupt Length)
XMTFLUSH (Transmit Buffer Flush)
RCVFLUSH (Receive Buffer Flush)

Figure 23-22. TWI FIFO Control Register

Additional information for the TWI_FIFO_CTL register bits includes:

- **Receive buffer interrupt length** (RCVINTLEN)

  This bit determines the rate at which receive buffer interrupts are to be generated. Interrupts may be generated with each byte received or after two bytes are received.

  [0] An interrupt (RCVSERV) is set when RCVSTAT indicates one or two bytes in the FIFO are full (01 or 11).

  [1] An interrupt (RCVSERV) is set when the RCVSTAT field in the TWI_FIFO_STAT register indicates two bytes in the FIFO are full (11).

- **Transmit buffer interrupt length** (XMTINTLEN)

  This bit determines the rate at which transmit buffer interrupts are to be generated. Interrupts may be generated with each byte transmitted or after two bytes are transmitted.

  [0] An interrupt (XMTSERV) is set when XMTSTAT indicates one or two bytes in the FIFO are empty (01 or 00).

  [1] An interrupt (XMTSERV) is set when the XMTSTAT field in the TWI_FIFO_STAT register indicates two bytes in the FIFO are empty (00).

- **Receive buffer flush** (RCVFLUSH)

  [0] Normal operation of the receive buffer and its status bits.

  [1] Flush the contents of the receive buffer and update the RCVSTAT status bit to indicate the buffer is empty. This state is held until this bit is cleared. During an active receive the receive buffer in this state responds to the receive logic as if it is full.

- **Transmit buffer flush** (XMTFLUSH)

[0] Normal operation of the transmit buffer and its status bits.

[1] Flush the contents of the transmit buffer and update the XMTSTAT status bit to indicate the buffer is empty. This state is held until this bit is cleared. During an active transmit the transmit buffer in this state responds as if the transmit buffer is empty.

TWI FIFO Status Register (TWI_FIFO_STAT)

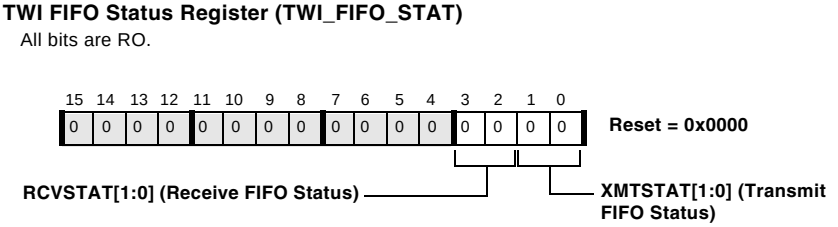

TWI FIFO Status Register (TWI_FIFO_STAT)
All bits are RO.
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RCVSTAT[1:0] (Receive FIFO Status)
XMTSTAT[1:0] (Transmit FIFO Status)

Figure 23-23. TWI FIFO Status Register

TWI FIFO Status

The fields in the TWI_FIFO_STAT register indicate the state of the FIFO buffers' receive and transmit contents. The FIFO buffers do not discriminate between master data and slave data. By using the status and control bits provided, the FIFO can be managed to allow simultaneous master and slave operation.

- **Receive FIFO status** (RCVSTAT[1:0])

  The RCVSTAT field is read only. It indicates the number of valid data bytes in the receive FIFO buffer. The status is updated with each FIFO buffer read using the peripheral data bus or write access by the receive shift register. Simultaneous accesses are allowed.

  [00] The FIFO is empty.

[01] The FIFO contains one byte of data. A single byte peripheral read of the FIFO is allowed.

[10] Reserved

[11] The FIFO is full and contains two bytes of data. Either a single or double byte peripheral read of the FIFO is allowed.

- **Transmit FIFO status** (`XMTSTAT[1:0]`)

  The `XMTSTAT` field is read only. It indicates the number of valid data bytes in the FIFO buffer. The status is updated with each FIFO buffer write using the peripheral data bus or read access by the transmit shift register. Simultaneous accesses are allowed.

  [00] The FIFO is empty. Either a single or double byte peripheral write of the FIFO is allowed.

  [01] The FIFO contains one byte of data. A single byte peripheral write of the FIFO is allowed.

  [10] Reserved

  [11] The FIFO is full and contains two bytes of data.

TWI Interrupt Mask Register (TWI_INT_MASK)

The `TWI_INT_MASK` register enables interrupt sources to assert the interrupt output. Each mask bit corresponds with one interrupt source bit in the `TWI_INT_STAT` register. Reading and writing the `TWI_INT_MASK` register does not affect the contents of the `TWI_INT_STAT` register.

TWI Interrupt Mask Register (TWI_INT_MASK)
For all bits, 0 = Interrupt generation disabled, 1 = Interrupt generation enabled.

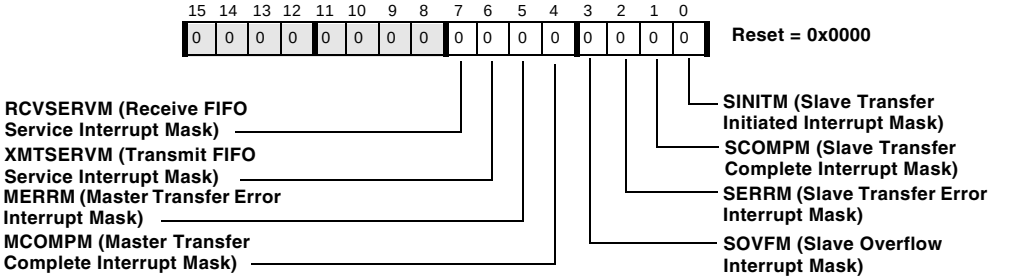

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RCVSERVM (Receive FIFO Service Interrupt Mask)
XMTSERVM (Transmit FIFO Service Interrupt Mask)
MERRM (Master Transfer Error Interrupt Mask)
MCOMPM (Master Transfer Complete Interrupt Mask)
SINITM (Slave Transfer Initiated Interrupt Mask)
SCOMPM (Slave Transfer Complete Interrupt Mask)
SERRM (Slave Transfer Error Interrupt Mask)
SOVFM (Slave Overflow Interrupt Mask)

Figure 23-24. TWI Interrupt Mask Register

TWI Interrupt Status Register (TWI_INT_STAT)

TWI Interrupt Status Register (TWI_INT_STAT)
All bits are sticky and W1C.

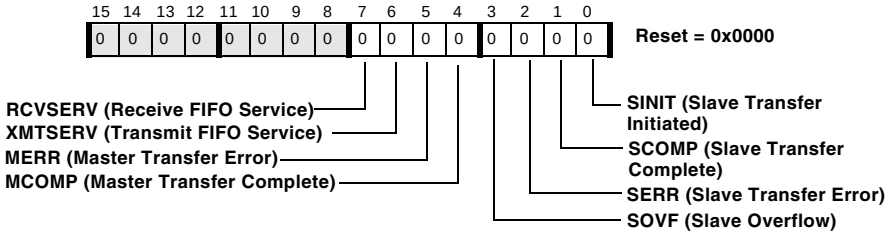

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RCVSERV (Receive FIFO Service)
XMTSERV (Transmit FIFO Service)
MERR (Master Transfer Error)
MCOMP (Master Transfer Complete)
SINIT (Slave Transfer Initiated)
SCOMP (Slave Transfer Complete)
SERR (Slave Transfer Error)
SOVF (Slave Overflow)

Figure 23-25. TWI Interrupt Status Register

The TWI_INT_STAT register contains information about functional areas requiring servicing. Many of the bits serve as an indicator to further read and service various status registers. After servicing the interrupt source

associated with a bit, the user must clear that interrupt source bit by writing a 1 to it.

- **Receive FIFO service** (RCVSERV)

  If RCVINTLEN in the TWI_FIFO_CTL register is 0, this bit is set each time the RCVSTAT field in the TWI_FIFO_STAT register is updated to either 01 or 11. If RCVINTLEN is 1, this bit is set each time RCVSTAT is updated to or 11.

  [0] No errors have been detected.

  [1] The FIFO does not require servicing or the RCVSTAT field has not changed since this bit was last cleared.

- **Transmit FIFO service** (XMTSERV)

  If XMTINTLEN in the TWI_FIFO_CTL register is 0, this bit is set each time the XMTSTAT field in the TWI_FIFO_STAT register is updated to either 01 or 00. If XMTINTLEN is 1, this bit is set each time XMTSTAT is updated to 00.

  [0] FIFO does not require servicing or XMTSTAT field has not changed since this bit was last cleared.

  [1] The transmit FIFO buffer has one or two 8-bit locations available to be written.

- **Master transfer error** (MERR)

  [0] No errors have been detected.

  [1] A master error has occurred. The conditions surrounding the error are indicated by the master status register (TWI_MASTER_STAT).

- **Master transfer complete** (MCOMP)

  [0] The completion of a transfer has not been detected.

[1] The initiated master transfer has completed. In the absence of a repeat start, the bus has been released.

- **Slave overflow** (SOVF)

  [0] No overflow has been detected.

  [1] The slave transfer complete (SCOMP) bit was set at the time a subsequent transfer has acknowledged an address phase. The transfer continues, however, it may be difficult to delineate data of one transfer from another.

- **Slave transfer error** (SERR)

  [0] No errors have been detected.

  [1] A slave error has occurred. A restart or stop condition has occurred during the data receive phase of a transfer.

- **Slave transfer complete** (SCOMP)

  [0] The completion of a transfer has not been detected.

  [1] The transfer is complete and either a stop, or a restart was detected.

- **Slave transfer initiated** (SINIT)

  [0] A transfer is not in progress. An address match has not occurred since the last time this bit was cleared.

  [1] The slave has detected an address match and a transfer has been initiated.

TWI FIFO Transmit Data Single Byte Register (TWI_XMT_DATA8)

The TWI_XMT_DATA8 register holds an 8-bit data value written into the FIFO buffer. Transmit data is entered into the corresponding transmit buffer in a first-in first-out order. For 16-bit PAB writes, a write access to TWI_XMT_DATA8 adds only one transmit data byte to the FIFO buffer. With each access, the transmit status (XMTSTAT) field in the TWI_FIFO_STAT register is updated. If an access is performed while the FIFO buffer is full, the write is ignored and the existing FIFO buffer data and its status remains unchanged.

TWI FIFO Transmit Data Single Byte Register (TWI_XMT_DATA8)
All bits are WO.

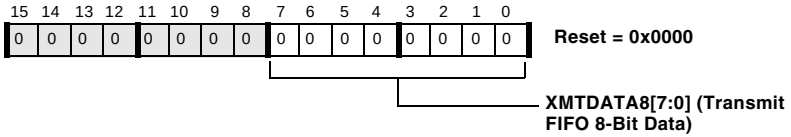


Figure 23-26. TWI FIFO Transmit Data Single Byte Register

TWI FIFO Transmit Data Double Byte Register (TWI_XMT_DATA16)

The TWI_XMT_DATA16 register holds a 16-bit data value written into the FIFO buffer. To reduce interrupt output rates and peripheral bus access times, a double byte transfer data access can be performed. Two data bytes can be written, effectively filling the transmit FIFO buffer with a single access.

The data is written in little endian byte order as shown in Figure 23-27 where byte 0 is the first byte to be transferred and byte 1 is the second byte to be transferred. With each access, the transmit status (XMTSTAT) field in the TWI_FIFO_STAT register is updated. If an access is performed while the

FIFO buffer is not empty, the write is ignored and the existing FIFO buffer data and its status remains unchanged.

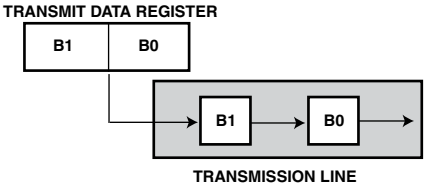


Figure 23-27. Transmit Little Endian Byte Order

TWI FIFO Transmit Data Double Byte Register (TWI_XMT_DATA16)
All bits are WO. This register always reads as 0x0000.

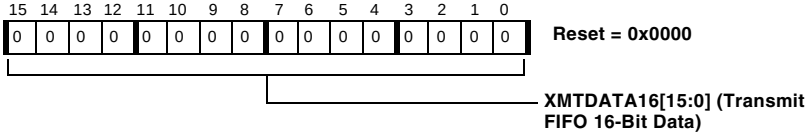


Figure 23-28. TWI FIFO Transmit Data Double Byte Register

TWI FIFO Receive Data Single Byte Register (TWI_RCV_DATA8)

The TWI_RCV_DATA8 register holds an 8-bit data value read from the FIFO buffer. Receive data is read from the corresponding receive buffer in a first-in first-out order. Although peripheral bus reads are 16 bits, a read access to TWI_RCV_DATA8 will access only one transmit data byte from the FIFO buffer. With each access, the receive status (RCVSTAT) field in the TWI_FIFO_STAT register is updated. If an access is performed while the

FIFO buffer is empty, the data is unknown and the FIFO buffer status remains indicating it is empty.

TWI FIFO Receive Data Single Byte Register (TWI_RCV_DATA8)
All bits are RO.

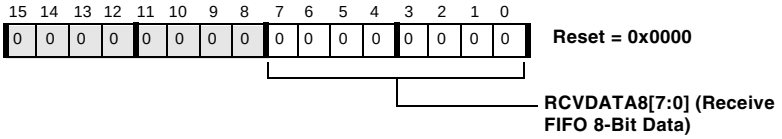


Figure 23-29. TWI FIFO Receive Data Single Byte Register

TWI FIFO Receive Data Double Byte Register (TWI_RCV_DATA16)

The TWI_RCV_DATA16 register holds a 16-bit data value read from the FIFO buffer. To reduce interrupt output rates and peripheral bus access times, a double byte receive data access can be performed. Two data bytes can be read, effectively emptying the receive FIFO buffer with a single access.

The data is read in little endian byte order as shown in Figure 23-30 where byte 0 is the first byte received and byte 1 is the second byte received. With each access, the receive status (RCVSTAT) field in the TWI_FIFO_STAT register is updated to indicate it is empty. If an access is

performed while the FIFO buffer is not full, the read data is unknown and the existing FIFO buffer data and its status remains unchanged.

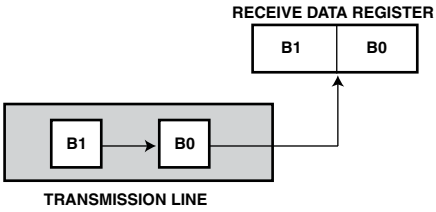


Figure 23-30. Receive Little Endian Byte Order

TWI FIFO Receive Data Double Byte Register (TWI_RCV_DATA16)
All bits are WO.

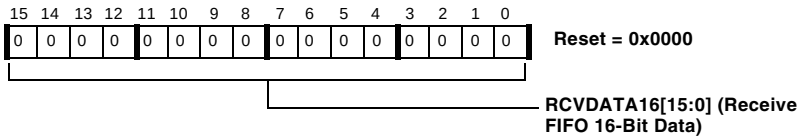


Figure 23-31. TWI FIFO Receive Data Double Byte Register

Programming Examples

The following sections include programming examples for general setup, slave mode, and master mode, as well as guidance for repeated start conditions.

Master Mode Setup

Listing 23-1 shows how to initiate polled receive and transmit transfers in master mode.

Listing 23-1. Master Mode Receive/Transmit Transfer

```
/*************************************************************
 Macro for the count field of the TWI_MASTER_CTL register
 x can be any value between 0 and 0xFE (254). A value of
 0xFF disables the counter.
*************************************************************/
#define TWICount(x) (DCNT & ((x) << 6))

.section L1_data_b;
.byte TX_file[file_size] = "DATA.hex";
.BYTE RX_CHECK[file_size];
.byte rcvFirstWord[2];

.SECTION program;
_main:
/*************************************************************
TWI Master Initialization subroutine
*************************************************************/

TWI_INIT:
/*************************************************************
Enable the TWI controller and set the Prescale value
Prescale = 10 (0xA) for an SCLK = 100 MHz (CLKIN = 50MHz)
Prescale = SCLK / 10 MHz

P1 points to the base of the system MMRs
*************************************************************/

R1 = TWI_ENA | 0xA (z);
W[P1 + LO(TWI_CONTROL)] = R1;

/*************************************************************
Set CLKDIV:
```

```
For example, for an SCL of 400 KHz (period = 1/400 KHz = 2500 ns)
and an internal time reference of 10 MHz (period = 100 ns):
CLKDIV = 2500 ns / 100 ns = 25
For an SCL with a 30% duty cycle, then CLKLOW = 17 (0x11) and
CLKHI = 8.
*************************************************************/
R5 = CLKHI(0x8) | CLKLOW(0x11) (z);
W[P1 + LO(TWI_CLKDIV)] = R5;

/*************************************************************
enable these signals to generate a TWI interrupt: optional
*************************************************************/
R1 = RCVSERV | XMTSERV | MERR | MCOMP (z);
W[P1 + LO(TWI_INT_MASK)] = R1;

/*************************************************************
The address needs to be shifted one place to the right
e.g., 1010 001x becomes 0101 0001 (0x51) the TWI controller
will actually send out 1010 001x where x is either a 0 for
writes or 1 for reads
*************************************************************/
R6 = 0xBF;
R6 = R6 >> 1;
TWI_INIT.END: W[P1 + LO(TWI_MASTER_ADDR)] = R6;

/******************** END OF TWI INIT **********************/

/*************************************************************
Starting the Read transfer
Program the Master Control register with:
1. the number of bytes to transfer: TWICount(x)
2. Repeated Start (RESTART): optional
3. speed mode: FAST or SLOW
```

```
4. direction of transfer:
      MDIR = 1 for reads, MDIR = 0 for writes
5. Master Enable MEN. This will kick off the master transfer
************************************************************/
R1 = TWICount(0x2) | FAST | MDIR | MEN;
W[P1 + LO(TWI_MASTER_CTL)] = R1;
ssync;

/************************************************************
Poll the FIFO Status register to know when
2 bytes have been shifted into the RX FIFO
************************************************************/
Rx_stat:
R1 = W[P1 + LO(TWI_FIFO_STAT)](Z);
R0 = 0xC;
R1 = R1 & R0;
CC = R1 == R0;
IF ! cc jump Rx_stat;
R0 = W[P1 + LO(TWI_RCV_DATA16)](Z); /* Read data from the RX fifo
*/
ssync;

/************************************************************
check that master transfer has completed
MCOMP will be set when Count reaches zero
************************************************************/
M_COMP:
  R1 = W[P1 + LO(TWI_INT_STAT)](z);
  CC = BITTST (R1, bitpos(MCOMP));
  if ! CC jump M_COMP;
M_COMP.END:  W[P1 + LO(TWI_INT_STAT)] = R1;

/* load the pointer with the address of the transmit buffer */
P2.H = TX_file;
```

```
P2.L = TX_file;

/*********************************************************
Pre-load the tx FIFO with the first two bytes: this is
necessary to avoid the generation of the Buffer Read Error
(BUFRDERR) which occurs whenever a transmit transfer is
initiated while the transmit buffer is empty
*********************************************************/
R3 = W[P2++](Z);
W[P1 + LO(TWI_XMT_DATA16)] = R3;

/*********************************************************
Initiating the Write operation
Program the Master Control register with:
1. the number of bytes to transfer: TWICount(x)
2. Repeated Start (RESTART): optional
3. speed mode: FAST or Standard
4. direction of transfer:
      MDIR = 1 for reads, MDIR = 0 for writes
5. Master Enable MEN. Setting this bit will kick off the transfer
*********************************************************/
R1 = TWICount(0xFE) | FAST | MEN;
W[P1 + LO(TWI_MASTER_CTL)] = R1;
SSYNC;

/*********************************************************
  loop to write data to a TWI slave device P3 times
*********************************************************/
P3 = length(TX_file);

LSETUP (Loop_Start1, Loop_End1) LC0 = P3;
  Loop_Start1:
    /*******************************************************
    check that there's at least one byte location empty in
```

```
    the tx fifo
    ************************************************************/
    XMTSERV_Status:
    R1 = W[P1 + LO(TWI_INT_STAT)](z);
    CC = BITTST (R1, bitpos(XMTSERV)); /* test XMTSERV bit */
    if ! CC jump XMTSERV_Status;
    W[P1 + LO(TWI_INT_STAT)] = R1; /* clear status */
    SSYNC;

    /*********************************************************
    write byte into the transmit FIFO
    *********************************************************/
    R3 = B[P2++](Z);
    W[P1 + LO(TWI_XMT_DATA8)] = R3;
  Loop_End1:  SSYNC;

/* check that master transfer has completed */
M_COMP1:
R1 = W[P1 + LO(TWI_INT_STAT)](z);
CC = BITTST (R1, bitpos(MCOMP1));
if ! CC jump M_COMP;
M_COMP1.END:W[P1 + LO(TWI_INT_STAT)] = R1;

idle;
_main.end:
```

Slave Mode Setup

Listing 23-2 shows how to configure the slave for interrupt based transfers. The interrupts are serviced in the subroutine `_TWI_ISR` shown in Listing 23-3.

Listing 23-2. Slave Mode Setup

```
#include <defBF527.h>
/*BF527 is used here as an example—change as appropriate.*/
#include "startup.h"

#define file_size 254
#define SYSMMR_BASE 0xFFC00000
#define COREMMR_BASE 0xFFE00000

.GLOBAL _main;
.EXTERN _TWI_ISR;

.section L1_data_b;
.BYTE TWI_RX[file_size];
.BYTE TWI_TX[file_size] = "transmit.dat";

.section L1_code;
_main:

/*********************************************************
TWI Slave Initialization subroutine
*********************************************************/
TWI_SLAVE_INIT:

/*********************************************************
Enable the TWI controller and set the Prescale value
Prescale = 10 (0xA) for an SCLK = 100 MHz (CLKIN = 50MHz)
Prescale = SCLK / 10 MHz
P1 points to the base of the system MMRs
P0 points to the base of the core MMRs
*********************************************************/
R1 = TWI_ENA | 0xA (z);
```

```
W[P1 + LO(TWI_CONTROL)] = R1;

/*********************************************************
Slave address
program the address to which this slave will respond to.
this is an arbitrary 7-bit value
*********************************************************/
R1 = 0x5F;
W[P1 + LO(TWI_SLAVE_ADDR)] = R1;

/*********************************************************
Pre-load the TX FIFO with the first two bytes to be
transmitted in the event the slave is addressed and a transmit
is required
*********************************************************/
R3=0xB537(Z);
W[P1 + LO(TWI_XMT_DATA16)] = R3;

/*********************************************************
FIFO Control determines whether an interrupt is generated
for every byte transferred or for every two bytes.
A value of zero which is the default, allows for single byte
events to generate interrupts
*********************************************************/
R1 = 0;
 W[P1 + LO(TWI_FIFO_CTL)] = R1;

/*********************************************************
enable these signals to generate a TWI interrupt
*********************************************************/
R1 = RCVSERV | XMTSERV | SOVF | SERR | SCOMP | SINIT (z);
W[P1 + LO(TWI_INT_MASK)] = R1;

/*********************************************************
```

```
Enable the TWI Slave
Program the Slave Control register with:
1. Slave transmit data valid (STDVAL) set so that the contents of
the TX FIFO can be used by this slave when a master requests data
from it.
2. Slave Enable SEN to enable Slave functionality
************************************************************/
R1 = STDVAL | SEN;
W[P1 + LO(TWI_SLAVE_CTL)] = R1;
TWI_SLAVE_INIT.END:

P2.H = HI(TWI_RX);
P2.L = LO(TWI_RX);

P4.H = HI(TWI_TX);
P4.L = LO(TWI_TX);
/************************************************************
Remap the vector table pointer from the default __I10HANDLER
to the new _TWI_ISR interrupt service routine
************************************************************/
R1.H = HI(_TWI_ISR);
R1.L = LO(_TWI_ISR);
[P0 + LO(EVT10)] = R1;   /* note that P0 points to the base of
the core MMR registers */

/************************************************************
ENABLE TWI generate to interrupts at the system level
************************************************************/
R1 = [P1 + LO(SIC_IMASK)];
BITSET(R1,BITPOS(IRQ_TWI));
[P1 + LO(SIC_IMASK)] = R1;

/************************************************************
ENABLE TWI to generate interrupts at the core level
```

```
*************************************************************/
R1 = [P0 + LO(IMASK)];
BITSET(R1,BITPOS(EVT_IVG10));
[P0 + LO(IMASK)] = R1;

/*************************************************************
 wait for interrupts
*************************************************************/
idle;

_main.END:
```

Listing 23-3. TWI Slave Interrupt Service Routine

```
/*************************************************************
Function:  _ TWI_ISR
Description:  This ISR is executed when the TWI controller
detects a slave initiated transfer. After an interrupt is ser-
viced, its corresponding bit is cleared in the TWI_INT_STAT
register. This done by writing a 1 to the particular bit posi-
tion. All bits are write 1 to clear.
*************************************************************/
#include <defBF527.h>
/*BF527 is used here as an example—change as appropriate.*/

.GLOBAL _TWI_ISR;

.section L1_code;
_TWI_ISR:

/*************************************************************
read the source of the interrupt
*************************************************************/
```

```
R1 = W[P1 + LO(TWI_INT_STAT)](z);

/*********************************************************
Slave Transfer Initiated
*********************************************************/
CC = BITTST(R1, BITPOS(SINIT));
if ! CC JUMP RECEIVE;
R0 = SINIT (Z);
W[P1 + LO(TWI_INT_STAT)] = R0;  /* clear interrupt source bit */
ssync;

/*********************************************************
Receive service
*********************************************************/
RECEIVE:
CC = BITTST(R1, BITPOS(RCVSERV));
if ! CC JUMP TRANSMIT;
R0 = W[P1 + LO(TWI_RCV_DATA8)] (Z);   /* read data */
B[P2++] = R0;   /* store bytes into a buffer pointed to by P2 */
R0 = RCVSERV(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /*clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END; /* exit */

/*********************************************************
Transmit service
*********************************************************/
TRANSMIT:
CC = BITTST(R1, BITPOS(XMTSERV));
if ! CC JUMP SlaveError;
R0 = B[P4++](Z);
W[P1 + LO(TWI_XMT_DATA8)] = R0;
R0 = XMTSERV(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /* clear interrupt source bit */
```

```
ssync;
JUMP _TWI_ISR.END; /* exit */

/*********************************************************
slave transfer error
*********************************************************/
SlaveError:
CC = BITTST(R1, BITPOS(SERR));
if ! CC JUMP SlaveOverflow;
R0 = SERR(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /* clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END;   /* exit */

/*********************************************************
slave overflow
*********************************************************/
SlaveOverflow:
CC = BITTST(R1, BITPOS(SOVF));
if !CC JUMP SlaveTransferComplete;
R0 = SOVF(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /* clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END;   /* exit */

/*********************************************************
 slave transfer complete
*********************************************************/
SlaveTransferComplete:
CC = BITTST(R1, BITPOS(SCOMP));
if !CC JUMP _TWI_ISR.END;
R0 = SCOMP(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /* clear interrupt source bit */
ssync;
```

```
/* Transfer complete read receive FIFO buffer and set/clear sema-
phores etc.... */
R0 = W[P1 + LO(TWI_FIFO_STAT)](z);
CC = BITTST(R0,BITPOS(RCV_HALF));    /* BIT 2 indicates whether
there's a byte in the FIFO or not */
if !CC JUMP _TWI_ISR.END;
R0 = W[P1 + LO(TWI_RCV_DATA8)] (Z);  /* read data */
B[P2++] = R0;   /* store bytes into a buffer pointed to by P2 */

_TWI_ISR.END:RTI;
```

Electrical Specifications

All logic complies with the Electrical Specification outlined in the *Philips I^2C Bus Specification version 2.1* dated January 2000.

Unique Information for the ADSP-BF52x Processor

None.

24 SPORT CONTROLLER

This chapter describes the synchronous serial peripheral port (SPORT). Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF52x

For details regarding the number of SPORTs for the ADSP-BF52x product, please refer to the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet*.

For SPORT DMA channel assignments, refer to Table 6-7 on page 6-106 in Chapter 6, “Direct Memory Access”.

For SPORT interrupt vector assignments, refer to Table 5-3 on page 5-18 in Chapter 5, “System Interrupts”.

To determine how each of the SPORTs is multiplexed with other functional pins, refer to Table 9-2 on page 9-5 through Table 9-5 on page 9-9 in Chapter 9, “General-Purpose Ports”.

For a list of MMR addresses for each SPORT, refer to Appendix A, “System MMR Assignments”.

SPORT behavior for the ADSP-BF52x that differs from the general information in this chapter can be found at the end of this chapter in the

section “Unique Information for the ADSP-BF52x Processor” on page 24-78.

Overview

Unlike the SPI interface which has been designed for SPI-compatible communication only, the SPORT modules support a variety of serial data communication protocols, for example:

- A-law or µ-law companding according to G.711 specification
- Multichannel or time-division-multiplexed (TDM) modes
- Stereo audio I^2S mode
- H.100 telephony standard support

In addition to these standard protocols, the SPORT module provides modes to connect to standard peripheral devices, such as ADCs or codecs, without external glue logic. With support for high data rates, independent transmit and receive channels, and dual data paths, the SPORT interface is a perfect choice for direct serial interconnection between two or more processors in a multiprocessor system. Many processors provide compatible interfaces, including processors from Analog Devices and other manufacturers.

Each SPORT has its own set of control registers and data buffers.

Features

A SPORT can operate at up to ½ the system clock (SCLK) rate for an internally generated or external serial clock. The SPORT external clock must always be less than the SCLK frequency. Independent transmit and receive clocks provide greater flexibility for serial communications.

A SPORT offers these features and capabilities:

- Provides independent transmit and receive functions.
- Transfers serial data words from 3 to 32 bits in length, either MSB first or LSB first.
- Provides alternate framing and control for interfacing to I^2S serial devices, as well as other audio formats (for example, left-justified stereo serial data).
- Has FIFO plus double buffered data (both receive and transmit functions have a data buffer register and a shift register), providing additional time to service the SPORT.
- Provides two synchronous transmit and two synchronous receive data signals and buffers to double the total supported datastreams.
- Performs A-law and μ-law hardware companding on transmitted and received words. (See "Companding" on page 24-29 for more information.)
- Internally generates serial clock and frame sync signals in a wide range of frequencies or accepts clock and frame sync input from an external source.
- Operates with or without frame synchronization signals for each data word, with internally generated or externally generated frame signals, with active high or active low frame signals, and with either of two configurable pulse widths and frame signal timing.
- Performs interrupt-driven, single word transfers to and from on-chip memory under processor control.

- Provides direct memory access transfer to and from memory under DMA master control. DMA can be autobuffer-based (a repeated, identical range of transfers) or descriptor-based (individual or repeated ranges of transfers with differing DMA parameters).
- Has a multichannel mode for TDM interfaces. A SPORT can receive and transmit data selectively from a time-division-multiplexed serial bitstream on 128 contiguous channels from a stream of up to 1024 total channels. This mode can be useful as a network communication scheme for multiple processors. The 128 channels available to the processor can be selected to start at any channel location from 0 to 895 (= 1023 – 128). Note the multichannel select registers and the WSIZE register control which subset of the 128 channels within the active region can be accessed.

Interface Overview

A SPORT provides an I/O interface to a wide variety of peripheral serial devices. SPORTs provide synchronous serial data transfer only. Each SPORT has one group of signals (primary data, secondary data, clock, and frame sync) for transmit and a second set of signals for receive. The receive and transmit functions are programmed separately. A SPORT is a full duplex device, capable of simultaneous data transfer in both directions. A SPORT can be programmed for bit rate, frame sync, and number of bits per word by writing to memory-mapped registers.

Figure 24-1 on page 24-6 shows a simplified block diagram of a single SPORT. Data to be transmitted is written from an internal processor register to the SPORT_TX register via the peripheral bus. This data is optionally compressed by the hardware and automatically transferred to the TX shift register. The bits in the shift register are shifted out on the DTPRI/DTSEC pin, MSB first or LSB first, synchronous to the serial clock on the TSCLK pin. The receive portion of the SPORT accepts data from the DRPRI/DRSEC pin synchronous to the serial clock on the RSCLK pin. When an entire word

is received, the data is optionally expanded, then automatically transferred to the SPORT_RX register, and then into the RX FIFO where it is available to the processor. Table 24-1 shows the signals for each SPORT.

Table 24-1. SPORT Signals

Pin	Description
DTxPRI	Transmit Data Primary
DTxSEC	Transmit Data Secondary
TSCLKx	Transmit Clock
TFSx	Transmit Frame Sync
DRxPRI	Receive Data Primary
DRxSEC	Receive Data Secondary
RSCLKx	Receive Clock
RFSx	Receive Frame Sync

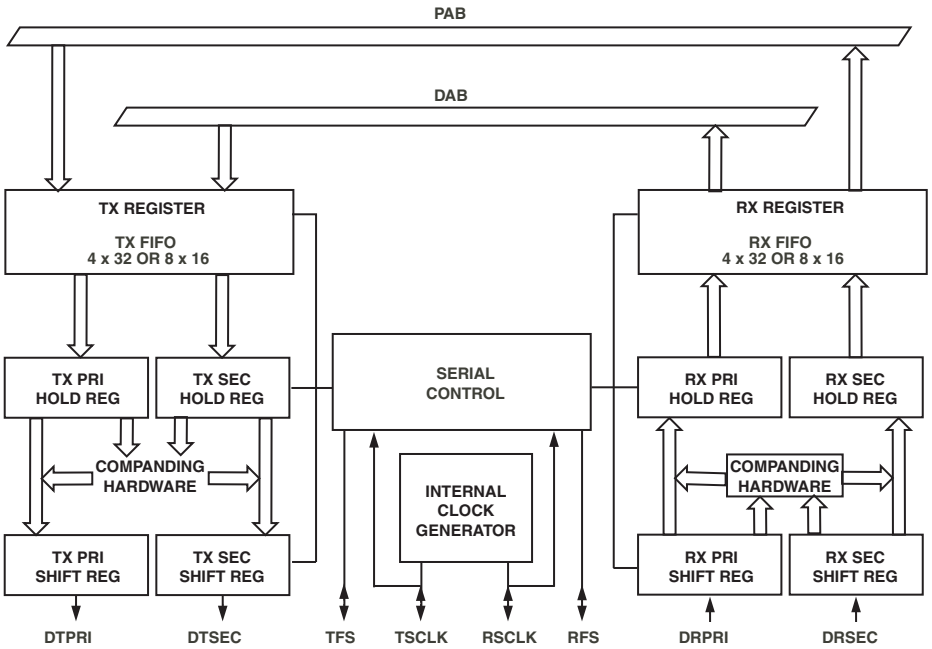


Figure 24-1. SPORT Block Diagram[1, 2, 3]

1 All wide arrow data paths are 16- or 32-bits wide, depending on SLEN. for SLEN = 2 to 15, a 16-bit data path with 8-deep fifo is used. for SLEN = 16 to 31, a 32-bit data path with 4-deep fifo is used.

2 TX register is the bottom of the TX fifo, RX register is the top of the RX fifo.

3 In multichannel mode, the TFS pin acts as transmit data valid (TDV). For more information, see "Multichannel Operation" on page 24-15.

A SPORT receives serial data on its DRPRI and DRSEC inputs and transmits serial data on its DTPRI and DTSEC outputs. It can receive and transmit simultaneously for full-duplex operation. For transmit, the data bits (DTPRI and DTSEC) are synchronous to the transmit clock (TSCLK). For receive, the data bits (DRPRI and DRSEC) are synchronous to the receive clock (RSCLK). The serial clock is an output if the processor generates it, or an input if the clock is externally generated. Frame synchronization signals RFS and TFS are used to indicate the start of a serial data word or stream of serial words.

The primary and secondary data pins, if enabled by a specific processor port configuration, provide a method to increase the data throughput of the serial port. They do not behave as totally separate SPORTs; rather, they operate in a synchronous manner (sharing clock and frame sync) but on separate data. The data received on the primary and secondary signals is interleaved in main memory and can be retrieved by setting a stride in the data address generators (DAG) unit. For more information about DAGs, see the *Data Address Generators* chapter in the Blackfin Processor Programming Reference. Similarly, for TX, data should be written to the TX register in an alternating manner—first primary, then secondary, then primary, then secondary, and so on. This is easily accomplished with the processor's powerful DAGs.

In addition to the serial clock signal, data must be signalled by a frame synchronization signal. The framing signal can occur either at the beginning of an individual word or at the beginning of a block of words.

Figure 24-2 on page 24-8 shows a possible port connection for a device with at least two SPORTs. Note serial devices A and B must be synchronous, as they share common frame syncs and clocks. The same is true for serial devices 1, 2, …N.

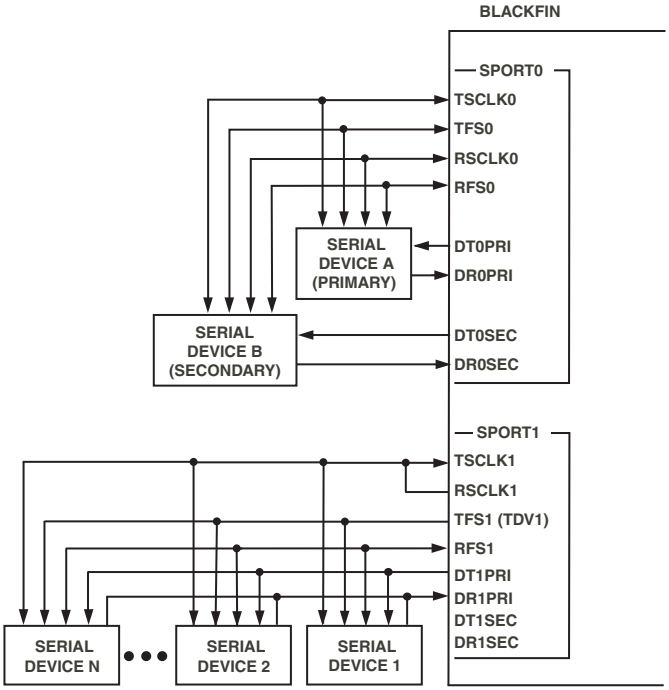


Figure 24-2. Example SPORT Connections (SPORT0 is Standard Mode, SPORT1 is Multichannel Mode)[1, 2]

1 In multichannel mode, TFS functions as a transmit data valid (TDV) output. See "Multichannel Operation" on page 24-15.

2 Although shown as an external connection, the TSCLK1/RSCLK1 connection is internal in multichannel mode. See "Multichannel Operation" on page 24-15.

Figure 24-3 shows an example of a stereo serial device with three transmit and two receive channels connected to a processor with two SPORTs.

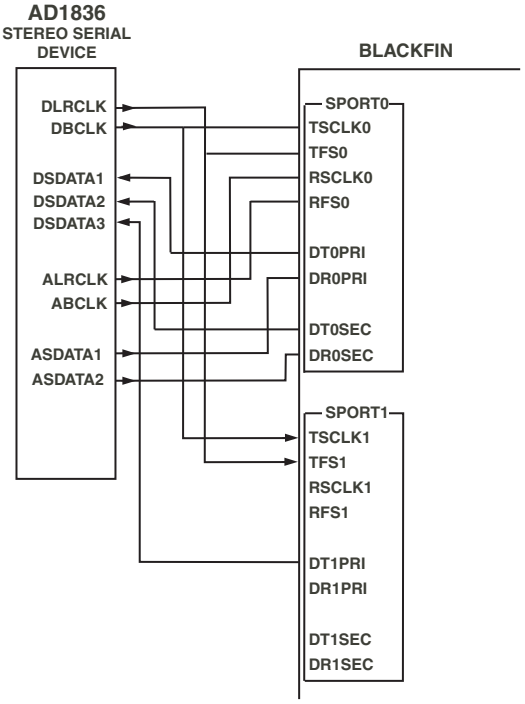


Figure 24-3. Stereo Serial Connection

SPORT Pin/Line Terminations

The processor has very fast drivers on all output pins, including the SPORTs. If connections on the data, clock, or frame sync lines are longer than six inches, consider using a series termination for strip lines on point-to-point connections. This may be necessary even when using low speed serial clocks, because of the edge rates.

Description of Operation

This section describes general SPORT operation, illustrating the most common use of a SPORT. Since the SPORT functionality is configurable, this description represents just one of many possible configurations.

Writing to a SPORT_TX register readies the SPORT for transmission. The TFS signal initiates the transmission of serial data. Once transmission has begun, each value written to the SPORT_TX register is transferred through the FIFO to the internal transmit shift register. The bits are then sent, beginning with either the MSB or the LSB as specified in the SPORT_TCR1 register. Each bit is shifted out on the driving edge of TSCLK. The driving edge of TSCLK can be configured to be rising or falling. The SPORT generates the transmit interrupt or requests a DMA transfer as long as there is space in the TX FIFO.

As a SPORT receives bits, they accumulate in an internal receive register. When a complete word has been received, it is written to the SPORT FIFO register and the receive interrupt for that SPORT is generated or a DMA transfer is initiated. Interrupts are generated differently if DMA block transfers are performed.

SPORT Disable

The SPORTs are automatically disabled by a processor hardware or software reset. A SPORT can also be disabled directly by clearing the SPORT's transmit or receive enable bits (TSPEN in the SPORT_TCR1 register and RSPEN in the SPORT_RCR1 register, respectively). Each method has a different effect on the SPORT.

A processor reset disables the SPORTs by clearing the SPORT_TCR1, SPORT_TCR2, SPORT_RCR1, and SPORT_RCR2 registers (including the TSPEN and RSPEN enable bits) and the SPORT_TCLKDIV, SPORT_RCLKDIV, SPORT_TFSDIVx, and SPORT_RFSDIVx clock and frame sync divisor registers. Any ongoing operations are aborted.

Clearing the TSPEN and RSPEN enable bits disables the SPORTs and aborts any ongoing operations. Status bits are also cleared. Configuration bits remain unaffected and can be read by the software in order to be altered or overwritten. To disable the SPORT output clock, set the SPORT to be disabled.

Note that disabling a SPORT via TSPEN/RSPEN may shorten any currently active pulses on the TFS/RFS and TSCLK/RSCLK outputs, if these signals are configured to be generated internally.

When disabling the SPORT from multichannel operation, first disable TXEN and then disable RXEN. Note both TXEN and RXEN must be disabled before re-enabling. Disabling only TX or RX is not allowed.

Setting SPORT Modes

SPORT configuration is accomplished by setting bit and field values in configuration registers. A SPORT must be configured prior to being enabled. Once the SPORT is enabled, further writes to the SPORT configuration registers are disabled (except for SPORT_RCLKDIV, SPORT_TCLKDIV, and multichannel mode channel select registers). To change values in all other SPORT configuration registers, disable the SPORT by clearing TSPEN in SPORT_TCR1 and/or RSPEN in SPORT_RCR1.

Each SPORT has its own set of control registers and data buffers. These registers are described in detail in "SPORT Registers" on page 24-46. All control and status bits in the SPORT registers are active high unless otherwise noted.

Stereo Serial Operation

Several stereo serial modes can be supported by the SPORT, including the popular I^2S format. To use these modes, set bits in the SPORT_RCR2 or SPORT_TCR2 registers. Setting RSFSE or TSFSE in SPORT_RCR2 or SPORT_TCR2 changes the operation of the frame sync pin to a left/right clock as

required for I^2S and left-justified stereo serial data. Setting this bit enables the SPORT to generate or accept the special LRCLK-style frame sync. All other SPORT control bits remain in effect and should be set appropriately. Figure 24-4 on page 24-14 and Figure 24-5 on page 24-15 show timing diagrams for stereo serial mode operation.

Table 24-2 on page 24-12 shows several modes that can be configured using bits in SPORT_TCR1 and SPORT_RCR1. The table shows bits for the receive side of the SPORT, but corresponding bits are available for configuring the transmit portion of the SPORT. A control field which may be either set or cleared depending on the user's needs, without changing the standard, is indicated by an "X."

Blackfin SPORTs are designed such that, in I^2S master mode, LRCLK is held at the last driven logic level and does not transition, to provide an edge, after the final data word is driven out. Therefore, while transmitting a fixed number of words to an I^2S receiver that expects an LRCLK edge to receive the incoming data word, the SPORT should send a dummy word after transmitting the fixed number of words. The transmission of this dummy word toggles LRCLK, generating an edge. Transmission of the dummy word is not required when the I^2S receiver is a Blackfin SPORT.

Table 24-2. Stereo Serial Settings

Bit Field	Stereo Audio Serial Scheme		
	I^2S	Left-Justified	DSP Mode
RSFSE	1	1	0
RRFST	0	0	0
LARFS	0	1	0
LRFS	0	1	0
RFSR	1	1	1
RCKFE	1	0	0

Table 24-2. Stereo Serial Settings (Continued)

Bit Field	Stereo Audio Serial Scheme		
	I^2S	Left-Justified	DSP Mode
SLEN	2 – 31	2 – 31	2 – 31
RLSBIT	0	0	0
RFSDIV (If internal FS is selected.)	2 – Max	2 – Max	2 – Max
RXSE (Secondary Enable is available for RX and TX.)	X	X	X

Note most bits shown as a 0 or 1 may be changed depending on the user's preference, creating many other "almost standard" modes of stereo serial operation. These modes may be of use in interfacing to codecs with slightly non-standard interfaces. The settings shown in Table 24-2 on page 24-12 provide glueless interfaces to many popular codecs.

Note RFSDIV or TFSDIV must still be greater than or equal to SLEN. For I^2S operation, RFSDIV or TFSDIV is usually 1/64 of the serial clock rate. With RSFSE set, the formulas to calculate frame sync period and frequency (discussed in "Clock and Frame Sync Frequencies" on page 24-26) still apply, but now refer to one half the period and twice the frequency. For instance, setting RFSDIV or TFSDIV = 31 produces an LRCLK that transitions every 32 serial clock cycles and has a period of 64 serial clock cycles.

The LRFS bit determines the polarity of the RFS or TFS frame sync pin for the channel that is considered a "right" channel. Thus, setting LRFS = 0 (meaning that it is an active high signal) indicates that the frame sync is high for the "right" channel, thus implying that it is low for the "left" channel. This is the default setting.

The RRFST and TRFST bits determine whether the first word received or transmitted is a left or a right channel. If the bit is set, the first word received or transmitted is a right channel. The default is to receive or transmit the left channel word first.

The secondary DRSEC and DTSEC pins are useful extensions of the SPORT which pair well with stereo serial mode. Multiple I^2S streams of data can be transmitted or received using a single SPORT. Note the primary and secondary pins are synchronous, as they share clock and LRCLK (frame sync) pins. The transmit and receive sides of the SPORT need not be synchronous, but may share a single clock in some designs. See Figure 24-3 on page 24-9, which shows multiple stereo serial connections being made between the processor and an AD1836 codec.

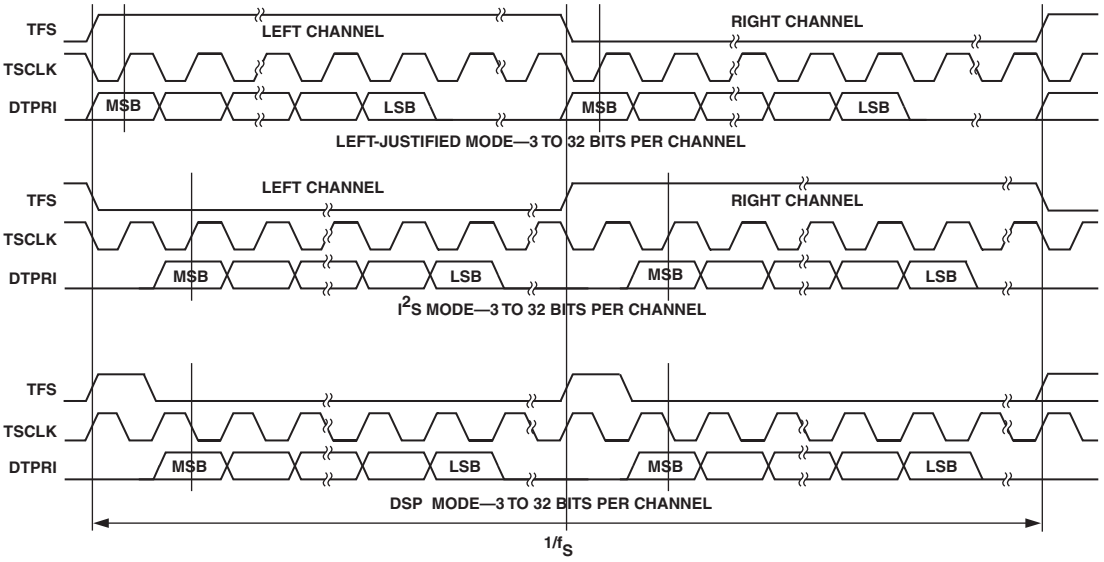


Figure 24-4. SPORT Stereo Serial Modes, Transmit[1, 2, 3]

1 DSP mode does not identify channel.
2 TFS normally operates at f_S except for DSP mode which is 2 x f_S.
3 TSCLK frequency is normally 64 x TFS but may be operated in burst mode.

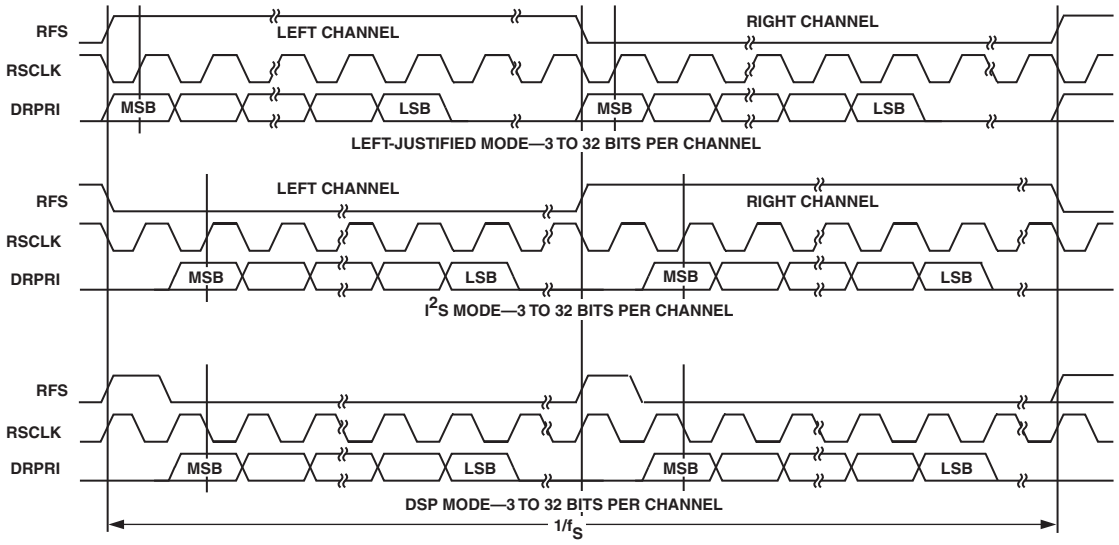


Figure 24-5. SPORT Stereo Serial Modes, Receive[1, 2, 3]

1 DSP mode does not identify channel.
2 RFS normally operates at f_S except for DSP mode which is 2 x f_S.
3 RSCLK frequency is normally 64 × RFS but may be operated in burst mode.

Multichannel Operation

The SPORT offers a multichannel mode of operation which allows the SPORT to communicate in a time-division-multiplexed (TDM) serial system. In multichannel communications, each data word of the serial bitstream occupies a separate channel. Each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of 24 channels.

The SPORT can automatically select words for particular channels while ignoring the others. Up to 128 channels are available for transmitting or receiving; each SPORT can receive and transmit data selectively from any of the 128 channels. These 128 channels can be any 128 out of the 1024

total channels. RX and TX must use the same 128-channel region to selectively enable channels. The SPORT can do any of the following on each channel:

- Transmit data
- Receive data
- Transmit and receive data
- Do nothing

Data companding and DMA transfers can also be used in multichannel mode.

The DTPRI pin is always driven (not three-stated) if the SPORT is enabled (TSPEN = 1 in the SPORT_TCR1 register), unless it is in multichannel mode and an inactive time slot occurs. The DTSEC pin is always driven (not three-stated) if the SPORT is enabled and the secondary transmit is enabled (TXSE = 1 in the SPORT_TCR2 register), unless the SPORT is in multichannel mode and an inactive time slot occurs.

The SPORT multichannel transmit select register and the SPORT multichannel receive select register must be programmed before enabling SPORT_TX or SPORT_RX operation for multichannel mode. This is especially important in "DMA data unpacked mode," since SPORT FIFO operation begins immediately after RSPEN and TSPEN are set, enabling both RX and TX. The MCMEN bit (in SPORT_MCMC2) must be enabled prior to enabling SPORT_TX or SPORT_RX operation. When disabling the SPORT from multichannel operation, first disable TXEN and then disable RXEN. Note both TXEN and RXEN must be disabled before re-enabling. Disabling only TX or RX is not allowed.

Figure 24-6 on page 24-17 shows example timing for a multichannel transfer that has these characteristics:

- Use TDM method where serial data is sent or received on different channels sharing the same serial bus
- Can independently select transmit and receive channels
- RFS signals start of frame
- TFS is used as "transmit data valid" for external logic, true only during transmit channels
- Receive on channels 0 and 2, transmit on channels 1 and 2
- Multichannel frame delay is set to 1

See "Timing Examples" on page 24-40 for more examples.

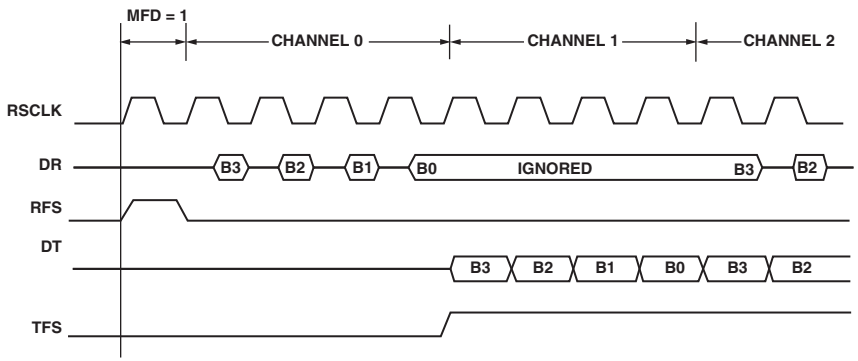


Figure 24-6. Multichannel Operation

Multichannel Enable

Setting the MCMEN bit in the SPORT_MCM2 register enables multichannel mode. When MCMEN = 1, multichannel operation is enabled; when MCMEN = 0, all multichannel operations are disabled.

Setting the MCMEN bit enables multichannel operation for *both* the receive and transmit sides of the SPORT. Therefore, if a receiving SPORT is in multichannel mode, the transmitting SPORT must also be in multichannel mode.

When in multichannel mode, do not enable the stereo serial frame sync modes or the late frame sync feature, as these features are incompatible with multichannel mode.

Table 24-3 shows the dependencies of bits in the SPORT configuration register when the SPORT is in multichannel mode.

Table 24-3. Multichannel Mode Configuration

SPORT_RCR1 or SPORT_RCR2	SPORT_TCR1 or SPORT_TCR2	Notes
RSPEN	TSPEN	Set or clear both
IRCLK	-	Independent
-	ITCLK	Independent
RDTYPE	TDTYPE	Independent
RLSBIT	TLSBIT	Independent
IRFS	-	Independent
-	ITFS	Ignored
RFSR	TFSR	Ignored
-	DITFS	Ignored
LRFS	LTFS	Independent
LARFS	LATFS	Both must be 0
RCKFE	TCKFE	Set or clear both to same value

Table 24-3. Multichannel Mode Configuration (Continued)

SPORT_RCR1 or SPORT_RCR2	SPORT_TCR1 or SPORT_TCR2	Notes
SLEN	SLEN	Set or clear both to same value
RXSE	TXSE	Independent
RSFSE	TSFSE	Both must be 0
RRFST	TRFST	Ignored

Frame Syncs in Multichannel Mode

All receiving and transmitting devices in a multichannel system must have the same timing reference. The RFS signal is used for this reference, indicating the start of a block or frame of multichannel data words.

When multichannel mode is enabled on a SPORT, both the transmitter and the receiver use RFS as a frame sync. This is true whether RFS is generated internally or externally. The RFS signal is used to synchronize the channels and restart each multichannel sequence. Assertion of RFS indicates the beginning of the channel 0 data word.

Since RFS is used by both the SPORT_TX and SPORT_RX channels of the SPORT in multichannel mode configuration, the corresponding bit pairs in SPORT_RCR1 and SPORT_TCR1, and in SPORT_RCR2 and SPORT_TCR2, should always be programmed identically, with the possible exception of the RXSE and TXSE pair and the RDTYPE and TDTYPE pair. This is true even if SPORT_RX operation is not enabled.

In multichannel mode, RFS timing similar to late (alternative) frame mode is entered automatically; the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted, provided that MFD is set to 0.

The TFS signal is used as a transmit data valid signal which is active during transmission of an enabled word. The SPORT's data transmit pin is three-stated when the time slot is not active, and the TFS signal serves as an

output-enabled signal for the data transmit pin. The SPORT drives TFS in multichannel mode whether or not ITFS is cleared. The TFS pin in multichannel mode still obeys the LTFS bit. If LTFS is set, the transmit data valid signal will be active low—a low signal on the TFS pin indicates an active channel.

Once the initial RFS is received, and a frame transfer has started, all other RFS signals are ignored by the SPORT until the complete frame has been transferred.

If MFD > 0, the RFS may occur during the last channels of a previous frame. This is acceptable, and the frame sync is not ignored as long as the delayed channel 0 starting point falls outside the complete frame.

In multichannel mode, the RFS signal is used for the block or frame start reference, after which the word transfers are performed continuously with no further RFS signals required. Therefore, internally generated frame syncs are always data independent.

The Multichannel Frame

A multichannel frame contains more than one channel, as specified by the window size and window offset. A complete multichannel frame consists of 1 – 1024 channels, starting with channel 0. The particular channels of the multichannel frame that are selected for the SPORT are a combination of the window offset, the window size, and the multichannel select registers. See Figure 24-7 on page 24-21.

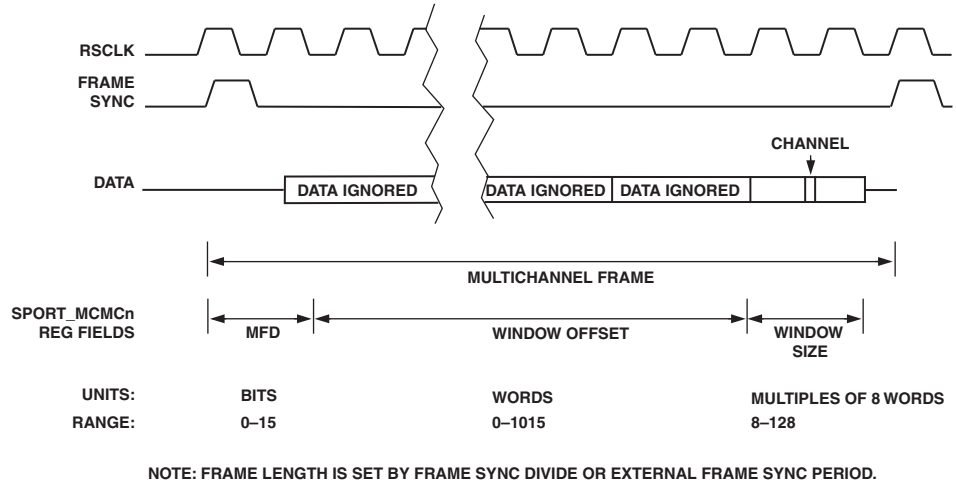


Figure 24-7. Relationships for Multichannel Parameters

Multichannel Frame Delay

The 4-bit MFD field in SPORT_MCMC2 specifies a delay between the frame sync pulse and the first data bit in multichannel mode. The value of MFD is the number of serial clock cycles of the delay. Multichannel frame delay allows the processor to work with different types of interface devices.

A value of 0 for MFD causes the frame sync to be concurrent with the first data bit. The maximum value allowed for MFD is 15. A new frame sync may occur before data from the last frame has been received, because blocks of data occur back-to-back.

Window Size

The window size (WSIZE[3:0]) defines the number of channels that can be enabled/disabled by the multichannel select registers. This range of words is called the active window. The number of channels can be any value in the range of 0 to 15, corresponding to active window size of 8 to 128, in increments of 8; the default value of 0 corresponds to a minimum active

window size of 8 channels. To calculate the active window size from the WSIZE register, use this equation:

Number of words in active window = 8 × (WSIZE + 1)

Since the DMA buffer size is always fixed, it is possible to define a smaller window size (for example, 32 words), resulting in a smaller DMA buffer size (in this example, 32 words instead of 128 words) to save DMA bandwidth. The window size cannot be changed while the SPORT is enabled.

Multichannel select bits that are enabled but fall outside the window selected are ignored.

Window Offset

The window offset (WOFF[9:0]) specifies where in the 1024-channel range to place the start of the active window. A value of 0 specifies no offset and 896 is the largest value that permits using all 128 channels. As an example, a program could define an active window with a window size of 8 (WSIZE = 0) and an offset of 93 (WOFF = 93). This 8-channel window would reside in the range from 93 to 100. Neither the window offset nor the window size can be changed while the SPORT is enabled.

If the combination of the window size and the window offset would place any portion of the window outside of the range of the channel counter, none of the out-of-range channels in the frame are enabled.

Other Multichannel Fields in SPORT_MCMC2

The FSDR bit in the SPORT_MCMC2 register changes the timing relationship between the frame sync and the clock received. This change enables the SPORT to comply with the H.100 protocol.

Normally (When FSDR = 0), the data is transmitted on the same edge that the TFS is generated. For example, a positive edge on TFS causes data to be transmitted on the positive edge of the TSCLK—either the same edge or the following one, depending on when LATFS is set.

When the frame sync/data relationship is used (FSDR = 1), the frame sync is expected to change on the falling edge of the clock and is sampled on the rising edge of the clock. This is true even though data received is sampled on the negative edge of the receive clock.

Channel Selection Register

A channel is a multibit word from 3 to 32 bits in length that belongs to one of the TDM channels. Specific channels can be individually enabled or disabled to select which words are received and transmitted during multichannel communications. Data words from the enabled channels are received or transmitted, while disabled channel words are ignored. Up to 128 contiguous channels may be selected out of 1024 available channels. The SPORT_MRCSn and SPORT_MTCSn multichannel select registers are used to enable and disable individual channels; the SPORT_MRCSn registers specify the active receive channels, and the SPORT_MTCSn registers specify the active transmit channels.

Four registers make up each multichannel select register. Each of the four registers has 32 bits, corresponding to 32 channels. Setting a bit enables that channel, so the SPORT selects its word from the multiple word block of data (for either receive or transmit). See Figure 24-8.

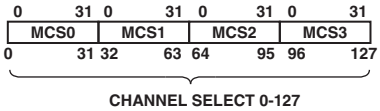


Figure 24-8. Multichannel Select Registers

Channel select bit 0 always corresponds to the first word of the active window. To determine a channel's absolute position in the frame, add the window offset words to the channel select position. For example, setting bit 7 in MCS2 selects word 71 of the active window to be enabled. Setting bit 2 in MCS1 selects word 34 of the active window, and so on.

Setting a particular bit in the SPORT_MTCSn register causes the SPORT to transmit the word in that channel's position of the datastream. Clearing the bit in the SPORT_MTCSn register causes the SPORT's data transmit pin to three-state during the time slot of that channel.

Setting a particular bit in the SPORT_MRCSn register causes the SPORT to receive the word in that channel's position of the datastream; the received word is loaded into the SPORT_RX buffer. Clearing the bit in the SPORT_MRCSn register causes the SPORT to ignore the data.

Companding may be selected for all channels or for no channels. A-law or μ-law companding is selected with the TDTYPE field in the SPORT_TCR1 register and the RDTYPE field in the SPORT_RCR1 register, and applies to all active channels. (See "Companding" on page 24-29 for more information about companding.)

Multichannel DMA Data Packing

Multichannel DMA data packing and unpacking are specified with the MCDTXPE and MCDRXPE bits in the SPORT_MCMC2 multichannel configuration register.

If the bits are set, indicating that data is packed, the SPORT expects the data contained by the DMA buffer corresponds only to the enabled SPORT channels. For example, if an MCM frame contains 10 enabled channels, the SPORT expects the DMA buffer to contain 10 consecutive words for each frame. It is not possible to change the total number of enabled channels without changing the DMA buffer size, and reconfiguration is not allowed while the SPORT is enabled.

If the bits are cleared (the default, indicating that data is not packed), the SPORT expects the DMA buffer to have a word for each of the channels in the active window, whether enabled or not, so the DMA buffer size must be equal to the size of the window. For example, if channels 1 and 10 are enabled, and the window size is 16, the DMA buffer size would have to be 16 words (unless the secondary side is enabled). The data to be

transmitted or received would be placed at addresses 1 and 10 of the buffer, and the rest of the words in the DMA buffer would be ignored. This mode allows changing the number of enabled channels while the SPORT is enabled, with some caution. First read the channel register to make sure that the active window is not being serviced. If the channel count is 0, then the multichannel select registers can be updated.

Support for H.100 Standard Protocol

The processor supports the H.100 standard protocol. The following SPORT parameters must be set to support this standard.

- Set for external frame sync. Frame sync generated by external bus master.
- TFSR/RFSR set (frame syncs required)
- LTFS/LRFS set (active low frame syncs)
- Set for external clock
- MCMEN set (multichannel mode selected)
- MFD = 0 (no frame delay between frame sync and first data bit)
- SLEN = 7 (8-bit words)
- FSDR = 1 (set for H.100 configuration, enabling half-clock-cycle early frame sync)

2× Clock Recovery Control

The SPORT can recover the data rate clock from a provided 2× input clock. This enables the implementation of H.100 compatibility modes for MVIP-90 (2 Mbps data) and HMVIP (8 Mbps data), by recovering 2 MHz from 4 MHz or 8 MHz from the 16 MHz incoming clock with the proper phase relationship. A 2-bit mode signal (MCCRM[1:0] in the

SPORT_MCMC2 register) chooses the applicable clock mode, which includes a non-divide or bypass mode for normal operation. A value of MCCRM = 00 chooses non-divide or bypass mode (H.100-compatible), MCCRM = 10 chooses MVIP-90 clock divide (extract 2 MHz from 4 MHz), and MCCRM = 11 chooses HMVIP clock divide (extract 8 MHz from 16 MHz).

Functional Description

The following sections provide a functional description of the SPORT.

Clock and Frame Sync Frequencies

The maximum serial clock frequency (for either an internal source or an external source) is SCLK/2. The frequency of an internally generated clock is a function of the system clock frequency (SCLK) and the value of the 16-bit serial clock divide modulus registers, SPORT_TCLKDIV and SPORT_RCLKDIV.

TSCLK frequency = (SCLK frequency)/(2 × (SPORT_TCLKDIV + 1))

RSCLK frequency = (SCLK frequency)/(2 × (SPORT_RCLKDIV + 1))

If the value of SPORT_TCLKDIV or SPORT_RCLKDIV is changed while the internal serial clock is enabled, the change in TSCLK or RSCLK frequency takes effect at the start of the drive edge of TSCLK or RSCLK that follows the next leading edge of TFS or RFS.

When an internal frame sync is selected (ITFS = 1 in the SPORT_TCR1 register or IRFS = 1 in the SPORT_RCR1 register) and frame syncs are not required, the first frame sync does not update the clock divider if the value in SPORT_TCLKDIV or SPORT_RCLKDIV has changed. The second frame sync will cause the update.

The SPORT_TFSDIV and SPORT_RFSDIV registers specify the number of transmit or receive clock cycles that are counted before generating a TFS or

RFS pulse (when the frame sync is internally generated). This enables a frame sync to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks.

The formula for the number of cycles between frame sync pulses is:

# of transmit serial clocks between frame sync assertions = TFSDIV + 1

# of receive serial clocks between frame sync assertions = RFSDIV + 1

Use the following equations to determine the correct value of TFSDIV or RFSDIV, given the serial clock frequency and desired frame sync frequency:

SPORT TFS frequency = (TSCLK frequency)/(SPORT_TFSDIV + 1)

SPORT RFS frequency = (RSCLK frequency)/(SPORT_RFSDIV + 1)

The frame sync would thus be continuously active (for transmit if TFSDIV = 0 or for receive if RFSDIV = 0). However, the value of TFSDIV (or RFSDIV) should not be less than the serial word length minus 1 (the value of the SLEN field in SPORT_TCR2 or SPORT_RCR2). A smaller value could cause an external device to abort the current operation or have other unpredictable results. If a SPORT is not being used, the TFSDIV (or RFSDIV) divisor can be used as a counter for dividing an external clock or for generating a periodic pulse or periodic interrupt. The SPORT must be enabled for this mode of operation to work.

Maximum Clock Rate Restrictions

Externally generated late transmit frame syncs also experience a delay from arrival to data output, and this can limit the maximum serial clock speed. See the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet* for exact timing specifications.

Word Length

Each SPORT channel (transmit and receive) independently handles word lengths of 3 to 32 bits. The data is right-justified in the SPORT data registers if it is fewer than 32 bits long, residing in the LSB positions. The value of the serial word length (SLEN) field in the SPORT_TCR2 and SPORT_RCR2 registers of each SPORT determines the word length according to this formula:

Serial Word Length = SLEN + 1

The SLEN value should not be set to 0 or 1; values from 2 to 31 are allowed. Continuous operation (when the last bit of the current word is immediately followed by the first bit of the next word) is restricted to word sizes of 4 or longer (so SLEN≥ 3).

Bit Order

Bit order determines whether the serial word is transmitted MSB first or LSB first. Bit order is selected by the RLSBIT and TLSBIT bits in the SPORT_RCR1 and SPORT_TCR1 registers. When RLSBIT (or TLSBIT) = 0, serial words are received (or transmitted) MSB first. When RLSBIT (or TLSBIT) = 1, serial words are received (or transmitted) LSB first.

Data Type

The TDTYPE field of the SPORT_TCR1 register and the RDTYPE field of the SPORT_RCR1 register specify one of four data formats for both single and multichannel operation. See Table 24-4 on page 24-29.

Table 24-4. TDTYPE, RDTYPE, and Data Formatting

TDTYPE or RDTYPE	SPORT_TCR1 Data Formatting	SPORT_RCR1 Data Formatting
00	Normal operation	Zero fill
01	Reserved	Sign extend
10	Compand using μ-law	Compand using μ-law
11	Compand using A-law	Compand using A-law

These formats are applied to serial data words loaded into the SPORT_RX and SPORT_TX buffers. SPORT_TX data words are not actually zero filled or sign extended, because only the significant bits are transmitted.

Companding

Companding (a contraction of COMpressing and exPANDing) is the process of logarithmically encoding and decoding data to minimize the number of bits that must be sent. The SPORT supports the two most widely used companding algorithms, μ-law and A-law. The processor compands data according to the CCITT G.711 specification. The type of companding can be selected independently for each SPORT.

When companding is enabled, valid data in the SPORT_RX register is the right-justified, expanded value of the eight LSBs received and sign extended to 16 bits. A write to SPORT_TX causes the 16-bit value to be compressed to eight LSBs (sign extended to the width of the transmit word) and written to the internal transmit register. Although the companding standards support only 13-bit (A-law) or 14-bit (μ-law)

maximum word lengths, up to 16-bit word lengths can be used. If the magnitude of the word value is greater than the maximum allowed, the value is automatically compressed to the maximum positive or negative value.

Lengths greater than 16 bits are not supported for companding operation.

Clock Signal Options

Each SPORT has a transmit clock signal (TSCLK) and a receive clock signal (RSCLK). The clock signals are configured by the TCKFE and RCKFE bits of the SPORT_TCR1 and SPORT_RCR1 registers. Serial clock frequency is configured in the SPORT_TCLKDIV and SPORT_RCLKDIV registers.

The receive clock pin may be tied to the transmit clock if a single clock is desired for both receive and transmit.

Both transmit and receive clocks can be independently generated internally or input from an external source. The ITCLK bit of the SPORT_TCR1 configuration register and the IRCLK bit in the SPORT_RCR1 configuration register determines the clock source.

When IRCLK or ITCLK = 1, the clock signal is generated internally by the processor, and the TSCLK or RSCLK pin is an output. The clock frequency is determined by the value of the serial clock divisor in the SPORT_RCLKDIV register.

When IRCLK or ITCLK = 0, the clock signal is accepted as an input on the TSCLK or RSCLK pins, and the serial clock divisors in the SPORT_TCLKDIV/SPORT_RCLKDIV registers are ignored. The externally generated serial clocks do not need to be synchronous with the system clock or

with each other. The system clock must have a higher frequency than RSCLK and TSCLK.

When the SPORT uses external clocks, it must be enabled for a minimal number of stable clock pulses before the first active frame sync is sampled. Failure to allow for these clocks may result in a SPORT malfunction. See the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet* for details.

The first internal frame sync will occur one frame sync delay after the SPORTs are ready. External frame syncs can occur as soon as the SPORT is ready.

Frame Sync Options

Framing signals indicate the beginning of each serial word transfer. The framing signals for each SPORT are TFS (transmit frame sync) and RFS (receive frame sync). A variety of framing options are available; these options are configured in the SPORT configuration registers (SPORT_TCR1, SPORT_TCR2, SPORT_RCR1 and SPORT_RCR2). The TFS and RFS signals of a SPORT are independent and are separately configured in the control registers.

Framed Versus Unframed

The use of multiple frame sync signals is optional in SPORT communications. The TFSR (transmit frame sync required select) and RFSR (receive frame sync required select) control bits determine whether frame sync signals are required. These bits are located in the SPORT_TCR1 and SPORT_RCR1 registers.

When TFSR = 1 or RFSR = 1, a frame sync signal is required for every data word. To allow continuous transmitting by the SPORT, each new data word must be loaded into the SPORT_TX hold register before the previous word is shifted out and transmitted.

When TFSR = 0 or RFSR = 0, the corresponding frame sync signal is not required. A single frame sync is needed to initiate communications but is ignored after the first bit is transferred. Data words are then transferred continuously, unframed.

With frame syncs not required, interrupt or DMA requests may not be serviced frequently enough to guarantee continuous unframed data flow. Monitor status bits or check for a SPORT Error interrupt to detect underflow or overflow of data.

Figure 24-9 on page 24-33 illustrates framed serial transfers, which have these characteristics:

- TFSR and RFSR bits in the SPORT_TCR1 and SPORT_RCR1 registers determine framed or unframed mode.
- Framed mode requires a framing signal for every word. Unframed mode ignores a framing signal after the first word.
- Unframed mode is appropriate for continuous reception.
- Active low or active high frame syncs are selected with the LTFS and LRFS bits of the SPORT_TCR1 and SPORT_RCR1 registers.

See "Timing Examples" on page 24-40 for more timing examples.

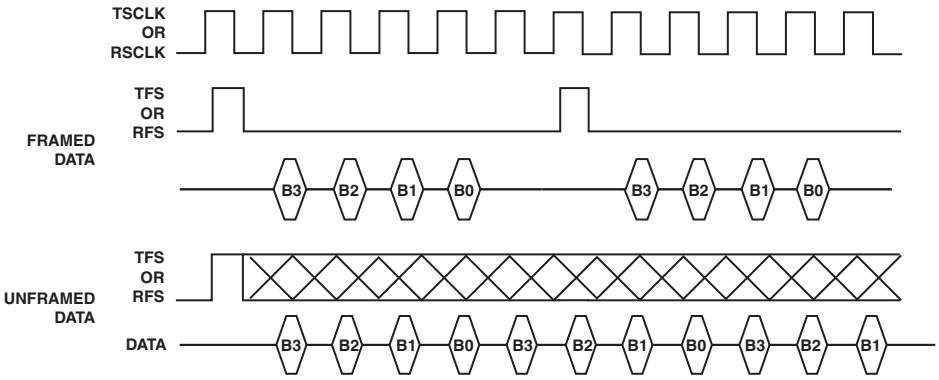


Figure 24-9. Framed Versus Unframed Data

Internal Versus External Frame Syncs

Both transmit and receive frame syncs can be independently generated internally or can be input from an external source. The ITFS and IRFS bits of the SPORT_TCR1 and SPORT_RCR1 registers determine the frame sync source.

When ITFS = 1 or IRFS = 1, the corresponding frame sync signal is generated internally by the SPORT, and the TFS pin or RFS pin is an output. The frequency of the frame sync signal is determined by the value of the frame sync divisor in the SPORT_TFSDIV or SPORT_RFSDIV register.

When ITFS = 0 or IRFS = 0, the corresponding frame sync signal is accepted as an input on the TFS pin or RFS pin, and the frame sync divisors in the SPORT_TFSDIV/SPORT_RFSDIV registers are ignored.

All of the frame sync options are available whether the signal is generated internally or externally.

Active Low Versus Active High Frame Syncs

Frame sync signals may be either active high or active low (in other words, inverted). The LTFS and LRFS bits of the SPORT_TCR1 and SPORT_RCR1 registers determine frame sync logic levels:

- When LTFS = 0 or LRFS = 0, the corresponding frame sync signal is active high.
- When LTFS = 1 or LRFS = 1, the corresponding frame sync signal is active low.

Active high frame syncs are the default. The LTFS and LRFS bits are initialized to 0 after a processor reset.

Sampling Edge for Data and Frame Syncs

Data and frame syncs can be sampled on either the rising or falling edges of the SPORT clock signals. The TCKFE and RCKFE bits of the SPORT_TCR1 and SPORT_RCR1 registers select the driving and sampling edges of the serial data and frame syncs.

For the SPORT transmitter, setting TCKFE = 1 in the SPORT_TCR1 register selects the falling edge of TSCLK to drive data and internally generated frame syncs and selects the rising edge of TSCLK to sample externally generated frame syncs. Setting TCKFE = 0 selects the rising edge of TSCLK to drive data and internally generated frame syncs and selects the falling edge of TSCLK to sample externally generated frame syncs.

For the SPORT receiver, setting RCKFE = 1 in the SPORT_RCR1 register selects the falling edge of RSCLK to drive internally generated frame syncs and selects the rising edge of RSCLK to sample data and externally generated frame syncs. Setting RCKFE = 0 selects the rising edge of RSCLK to drive internally generated frame syncs and selects the falling edge of RSCLK to sample data and externally generated frame syncs.

Note externally generated data and frame sync signals should change state on the opposite edge than that selected for sampling. For example, for an externally generated frame sync to be sampled on the rising edge of the clock (TCKFE = 1 in the SPORT_TCR1 register), the frame sync must be driven on the falling edge of the clock.

The transmit and receive functions of two SPORTs connected together should always select the same value for TCKFE in the transmitter and RCKFE in the receiver, so that the transmitter drives the data on one edge and the receiver samples the data on the opposite edge.

In Figure 24-10, TCKFE = RCKFE = 0 and transmit and receive are connected together to share the same clock and frame syncs.

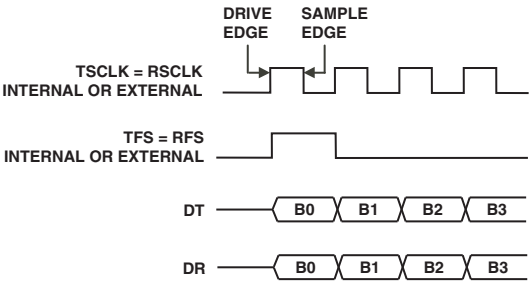


Figure 24-10. Example of TCKFE = RCKFE = 0, Transmit and Receive Connected

In Figure 24-11 on page 24-36, TCKFE = RCKFE = 1 and transmit and receive are connected together to share the same clock and frame syncs.

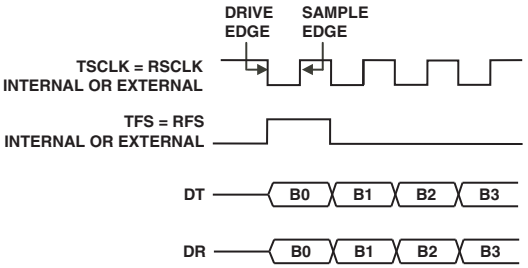


Figure 24-11. Example of TCKFE = RCKFE = 1, Transmit and Receive Connected

Early Versus Late Frame Syncs (Normal Versus Alternate Timing)

Frame sync signals can occur during the first bit of each data word (late) or during the serial clock cycle immediately preceding the first bit (early). The LATFS and LARFS bits of the SPORT_TCR1 and SPORT_RCR1 registers configure this option.

When LATFS = 0 or LARFS = 0, early frame syncs are configured; this is the normal mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the serial clock cycle after the frame sync is asserted, and the frame sync is not checked again until the entire word has been transmitted or received. In multichannel operation, this corresponds to the case when multichannel frame delay is 1.

If data transmission is continuous in early framing mode (in other words, the last bit of each word is immediately followed by the first bit of the next word), then the frame sync signal occurs during the last bit of each word. Internally generated frame syncs are asserted for one clock cycle in early framing mode. Continuous operation is restricted to word sizes of 4 or longer (SLEN $\geq$ 3).

When LATFS = 1 or LARFS = 1, late frame syncs are configured; this is the alternate mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted. In multichannel operation, this is the case when frame delay is 0. Receive data bits are sampled by serial clock edges, but the frame sync signal is only checked during the first bit of each word. Internally generated frame syncs remain asserted for the entire length of the data word in late framing mode. Externally generated frame syncs are only checked during the first bit.

Figure 24-12 on page 24-38 illustrates the two modes of frame signal timing. In summary:

- For the LATFS or LARFS bits of the SPORT_TCR1 or SPORT_RCR1 registers: LATFS = 0 or LARFS = 0 for early frame syncs, LATFS = 1 or LARFS = 1 for late frame syncs.
- For early framing, the frame sync precedes data by one cycle. For late framing, the frame sync is checked on the first bit only.
- Data is transmitted MSB first (TLSBIT = 0 or RLSBIT = 0) or LSB first (TLSBIT = 1 or RLSBIT = 1).
- Frame sync and clock are generated internally or externally.

See “Timing Examples” on page 24-40 for more examples.

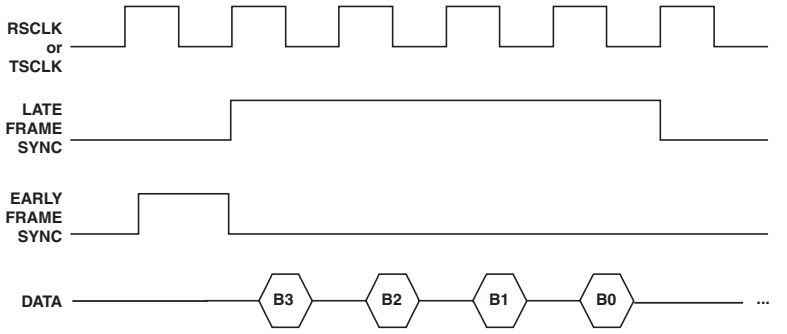


Figure 24-12. Normal Versus Alternate Framing

Data Independent Transmit Frame Sync

Normally the internally generated transmit frame sync signal (TFS) is output only when the SPORT_TX buffer has data ready to transmit. The data-independent transmit frame sync select bit (DITFS) allows the continuous generation of the TFS signal, with or without new data. The DITFS bit of the SPORT_TCR1 register configures this option.

When DITFS = 0, the internally generated TFS is only output when a new data word has been loaded into the SPORT_TX buffer. The next TFS is generated once data is loaded into SPORT_TX. This mode of operation allows data to be transmitted only when it is available.

When DITFS = 1, the internally generated TFS is output at its programmed interval regardless of whether new data is available in the SPORT_TX buffer. Whatever data is present in SPORT_TX is transmitted again with each assertion of TFS. The TUVF (transmit underflow status) bit in the SPORT_STAT register is set when this occurs and old data is retransmitted. The TUVF status bit is also set if the SPORT_TX buffer does not have new data when an externally generated TFS occurs. Note that in this mode of operation, data is transmitted only at specified times.

If the internally generated TFS is used, a single write to the SPORT_TX data register is required to start the transfer.

Moving Data Between SPORTs and Memory

Transmit and receive data can be transferred between the SPORTs and on-chip memory in one of two ways: with single word transfers or with DMA block transfers.

If no SPORT DMA channel is enabled, the SPORT generates an interrupt every time it has received a data word or needs a data word to transmit. SPORT DMA provides a mechanism for receiving or transmitting an entire block or multiple blocks of serial data before the interrupt is generated. The SPORT's DMA controller handles the DMA transfer, allowing the processor core to continue running until the entire block of data is transmitted or received. Interrupt service routines (ISRs) can then operate on the block of data rather than on single words, significantly reducing overhead.

SPORT RX, TX, and Error Interrupts

The SPORT RX interrupt is asserted when RSPEN is enabled and any words are present in the RX FIFO. If RX DMA is enabled, the SPORT RX interrupt is turned off and DMA services the RX FIFO.

The SPORT TX interrupt is asserted when TSPEN is enabled and the TX FIFO has room for words. If TX DMA is enabled, the SPORT TX interrupt is turned off and DMA services the TX FIFO.

The SPORT error interrupt is asserted when any of the sticky status bits (ROVF, RUVF, TOVF, TUVF) are set. The ROVF and RUVF bits are cleared by writing 0 to RSPEN. The TOVF and TUVF bits are cleared by writing 0 to TSPEN.

Peripheral Bus Errors

The SPORT generates a peripheral bus error for illegal register read or write operations. Examples include:

- Reading a write-only register (for example, SPORT_TX)
- Writing a read-only register (for example, SPORT_RX)
- Writing or reading a register with the wrong size (for example, 32-bit read of a 16-bit register)
- Accessing reserved register locations

Timing Examples

Several timing examples are included within the text of this chapter (in the sections "Framed Versus Unframed" on page 24-31, "Early Versus Late Frame Syncs (Normal Versus Alternate Timing)" on page 24-36, and "Frame Syncs in Multichannel Mode" on page 24-19). This section contains additional examples to illustrate other possible combinations of the framing options.

These timing examples show the relationships between the signals but are not scaled to show the actual timing parameters of the processor. Consult the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet* for actual timing parameters and values.

These examples assume a word length of four bits (SLEN = 3). Framing signals are active high (LRFS = 0 and LTFS = 0).

Figure 24-13 on page 24-41 through Figure 24-18 on page 24-43 show framing for receiving data.

In Figure 24-13 and Figure 24-14, the normal framing mode is shown for non-continuous data (any number of TSCLK or RSCLK cycles between words) and continuous data (no TSCLK or SCLK cycles between words).

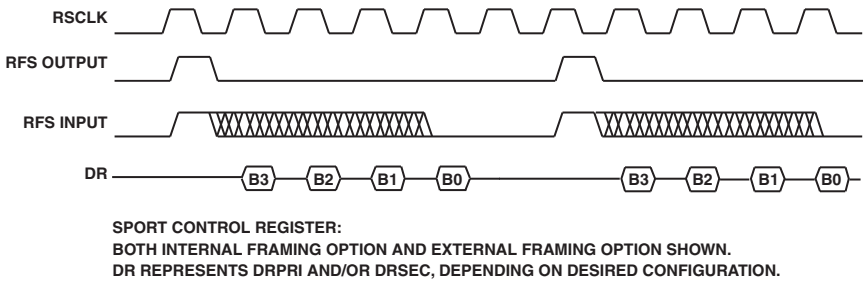


Figure 24-13. SPORT Receive, Normal Framing

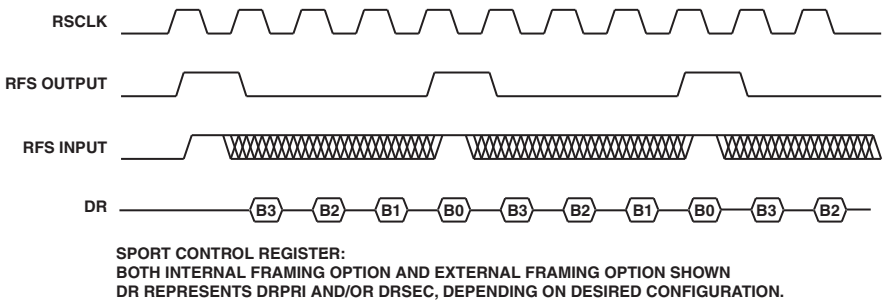


Figure 24-14. SPORT Continuous Receive, Normal Framing

Figure 24-15 on page 24-42 and Figure 24-16 on page 24-42 show non-continuous and continuous receiving in the alternate framing mode. These four figures show the input timing requirement for an externally generated frame sync and also the output timing characteristic of an internally generated frame sync. Note the output meets the input timing

requirement; therefore, with two SPORT channels used, one SPORT channel could provide RFS for the other SPORT channel.

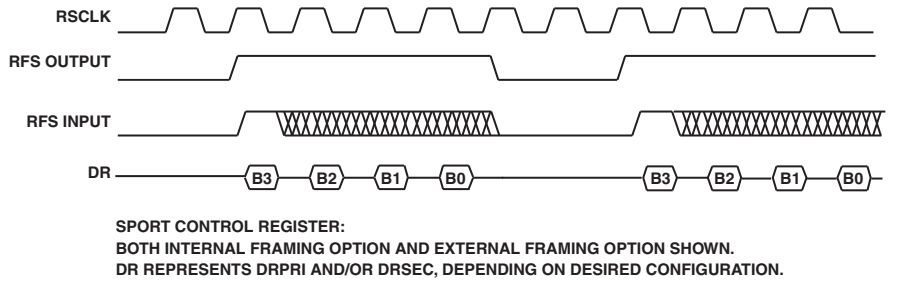


Figure 24-15. SPORT Receive, Alternate Framing

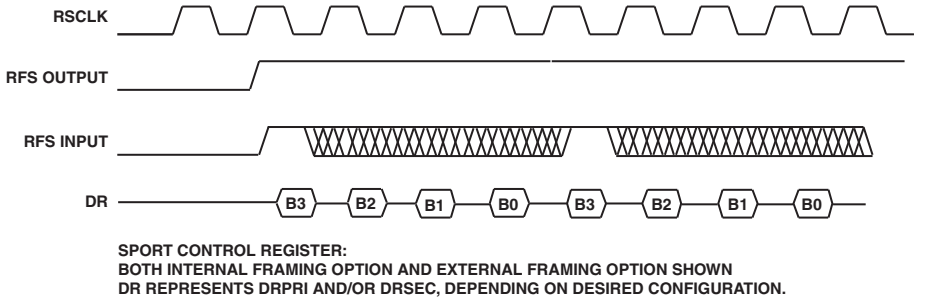


Figure 24-16. SPORT Continuous Receive, Alternate Framing

Figure 24-17 on page 24-43 and Figure 24-18 on page 24-43 show the receive operation with normal framing and alternate framing, respectively, in the unframed mode. A single frame sync signal occurs only at the start of the first word, either one RSCLK before the first bit (in normal mode) or

at the same time as the first bit (in alternate mode). This mode is appropriate for multiword bursts (continuous reception).

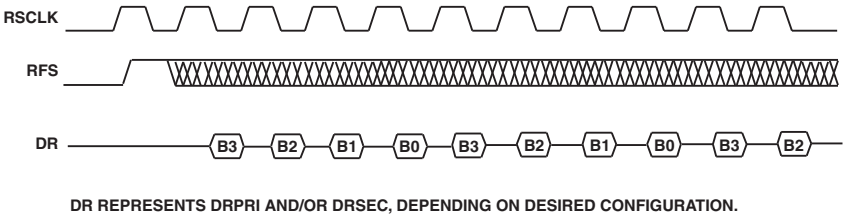


Figure 24-17. SPORT Receive, Unframed Mode, Normal Framing

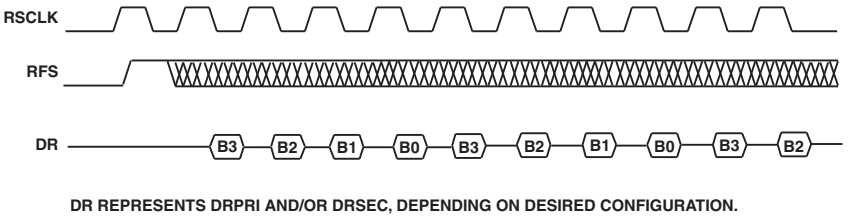


Figure 24-18. SPORT Receive, Unframed Mode, Alternate Framing

Figure 24-19 on page 24-44 through Figure 24-24 on page 24-46 show framing for transmitting data and are very similar to Figure 24-13 on page 24-41 through Figure 24-18 on page 24-43.

In Figure 24-19 on page 24-44 and Figure 24-20 on page 24-44, the normal framing mode is shown for non-continuous data (any number of TSCLK cycles between words) and continuous data (no TSCLK cycles between words). Figure 24-21 on page 24-45 and Figure 24-22 on page 24-45 show non-continuous and continuous transmission in the

alternate framing mode. As noted previously for the receive timing diagrams, the RFS output meets the RFS input timing requirement.

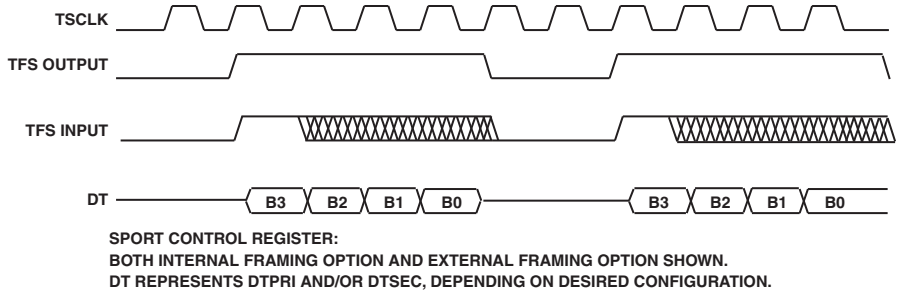


Figure 24-19. SPORT Transmit, Normal Framing

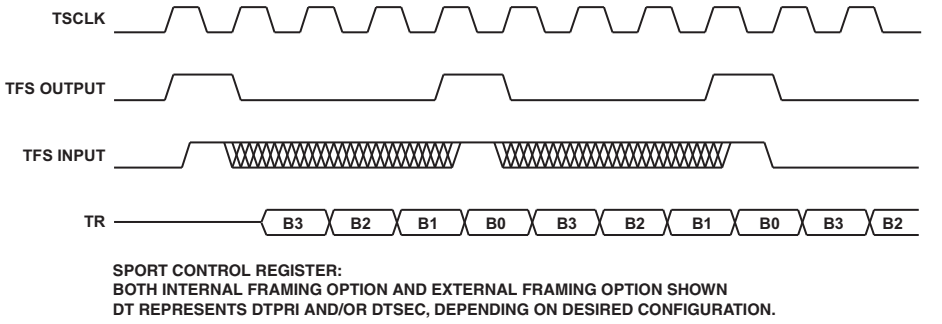


Figure 24-20. SPORT Continuous Transmit, Normal Framing

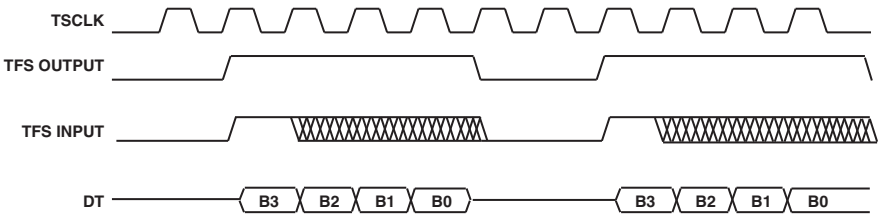


Figure 24-21. SPORT Transmit, Alternate Framing

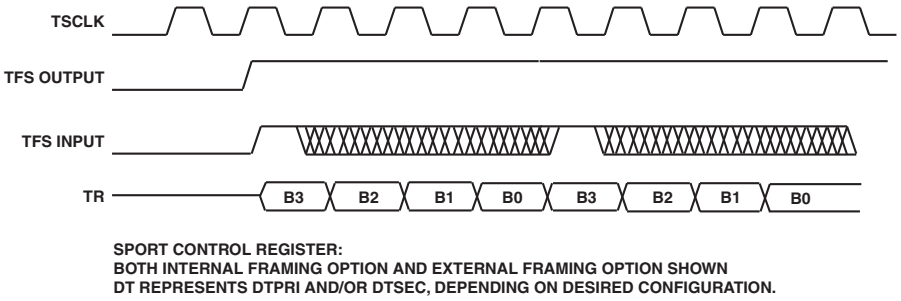


Figure 24-22. SPORT Continuous Transmit, Alternate Framing

Figure 24-23 on page 24-45 and Figure 24-24 on page 24-46 show the transmit operation with normal framing and alternate framing, respectively, in the unframed mode. A single frame sync signal occurs only at the start of the first word, either one TSCLK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode).

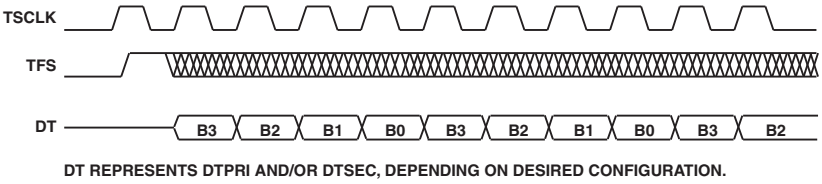


Figure 24-23. SPORT Transmit, Unframed Mode, Normal Framing

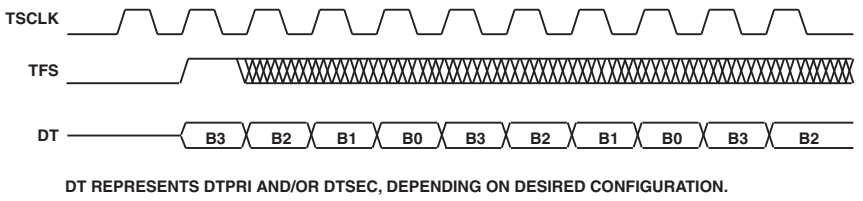


Figure 24-24. SPORT Transmit, Unframed Mode, Alternate Framing

SPORT Registers

The following sections describe the SPORT registers. Table 24-5 provides an overview of the available control registers.

Table 24-5. SPORT Register Mapping

Register Name	Function	Notes
SPORT_TCR1	Primary transmit configuration register	Bits [15:1] can only be written if bit 0 = 0
SPORT_TCR2	Secondary transmit configuration register	
SPORT_TCLKDIV	Transmit clock divider register	Ignored if external SPORT clock mode is selected
SPORT_TFSDIV	Transmit frame sync divider register	Ignored if external frame sync mode is selected
SPORT_TX	Transmit data register	See description of FIFO buffering at "SPORT Transmit Data (SPORT_TX) Register" on page 24-59
SPORT_RCR1	Primary receive configuration register	Bits [15:1] can only be written if bit 0 = 0
SPORT_RCR2	Secondary receive configuration register	
SPORT_RCLK_DIV	Receive clock divider register	Ignored if external SPORT clock mode is selected

Table 24-5. SPORT Register Mapping (Continued)

Register Name	Function	Notes
SPORT_RFSDIV	Receive frame sync divider register	Ignored if external frame sync mode is selected
SPORT_RX	Receive data register	See description of FIFO buffering at "SPORT Receive Data (SPORT_RX) Register" on page 24-61
SPORT_STAT	Receive and transmit status	
SPORT_MCM1	Primary multichannel mode configuration register	Configure this register before enabling the SPORT
SPORT_MCM2	Secondary multichannel mode configuration register	Configure this register before enabling the SPORT
SPORT_MRCSn	Receive channel selection registers	Select or deselect channels in a multichannel frame
SPORT_MTCSn	Transmit channel selection registers	Select or deselect channels in a multichannel frame
SPORT_CHNL	Currently serviced channel in a multichannel frame	

Register Writes and Effective Latency

When the SPORT is disabled (TSPEN and RSPEN cleared), SPORT register writes are internally completed at the end of the SCLK cycle in which they occurred, and the register reads back the newly-written value on the next cycle.

When the SPORT is enabled to transmit (TSPEN set) or receive (RSPEN set), corresponding SPORT configuration register writes are disabled (except for SPORT_RCLKDIV, SPORT_TCLKDIV, and multichannel mode channel select registers). The SPORT_TX register writes are always enabled; SPORT_RX, SPORT_CHNL, and SPORT_STAT are read-only registers.

After a write to a SPORT register, while the SPORT is disabled, any changes to the control and mode bits generally take effect when the SPORT is re-enabled.

Most configuration registers can only be changed while the SPORT is disabled (TSPEN/RSPEN = 0). Changes take effect after the SPORT is re-enabled. The only exceptions to this rule are the TCLKDIV/RCLKDIV registers and multichannel select registers.

SPORT Transmit Configuration (SPORT_TCR1 and SPORT_TCR2) Registers

The main control registers for the transmit portion of each SPORT are the transmit configuration registers, SPORT_TCR1 and SPORT_TCR2, shown in Figure 24-25 on page 24-49 and Figure 24-26 on page 24-50.

A SPORT is enabled for transmit if bit 0 (TSPEN) of the transmit configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT transmission.

When the SPORT is enabled to transmit (TSPEN set), corresponding SPORT configuration register writes are not allowed except for SPORT_TCLKDIV and multichannel mode channel select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORT_TCR1 is not written except for bit 0 (TSPEN). For example,

```
write (SPORT_TCR1, 0x0001) ;   /* SPORT TX Enabled */
write (SPORT_TCR1, 0xFF01) ;   /* ignored, no effect */
```

```
write (SPORT_TCR1, 0xFFF0) ;   /* SPORT disabled, SPORT_TCR1
                                  still equal to 0x0000 */
```

SPORT Transmit Configuration 1 Register (SPORT_TCR1)

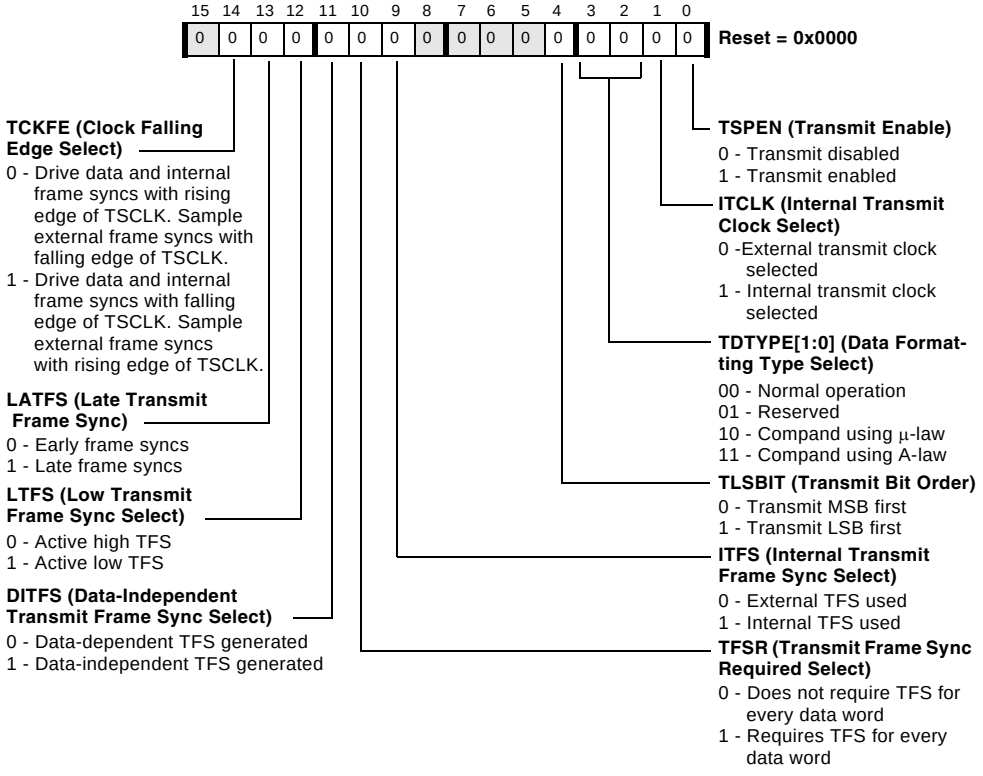

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
TCKFE (Clock Falling Edge Select)
0 - Drive data and internal frame syncs with rising edge of TSCLK. Sample external frame syncs with falling edge of TSCLK.
1 - Drive data and internal frame syncs with falling edge of TSCLK. Sample external frame syncs with rising edge of TSCLK.
LATFS (Late Transmit Frame Sync)
0 - Early frame syncs
1 - Late frame syncs
LTFS (Low Transmit Frame Sync Select)
0 - Active high TFS
1 - Active low TFS
DITFS (Data-Independent Transmit Frame Sync Select)
0 - Data-dependent TFS generated
1 - Data-independent TFS generated
TSPEN (Transmit Enable)
0 - Transmit disabled
1 - Transmit enabled
ITCLK (Internal Transmit Clock Select)
0 -External transmit clock selected
1 - Internal transmit clock selected
TDTYPE[1:0] (Data Formatting Type Select)
00 - Normal operation
01 - Reserved
10 - Compand using μ-law
11 - Compand using A-law
TLSBIT (Transmit Bit Order)
0 - Transmit MSB first
1 - Transmit LSB first
ITFS (Internal Transmit Frame Sync Select)
0 - External TFS used
1 - Internal TFS used
TFSR (Transmit Frame Sync Required Select)
0 - Does not require TFS for every data word
1 - Requires TFS for every data word

Figure 24-25. SPORT Transmit Configuration 1 Register

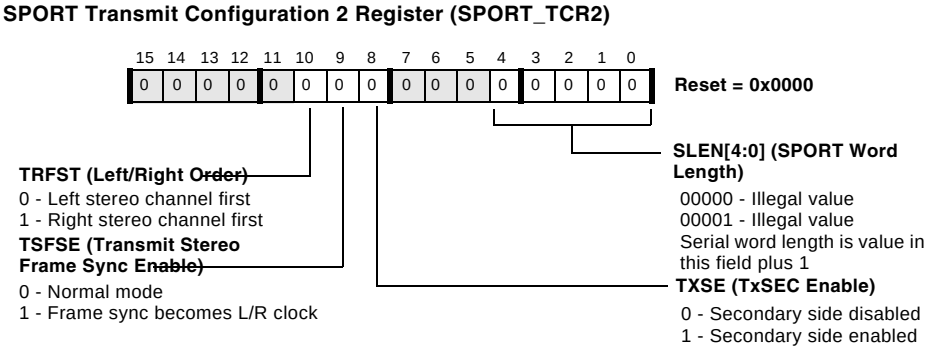

SPORT Transmit Configuration 2 Register (SPORT_TCR2)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SLEN[4:0] (SPORT Word Length)
00000 - Illegal value
00001 - Illegal value
Serial word length is value in this field plus 1
TRFST (Left/Right Order)
0 - Left stereo channel first
1 - Right stereo channel first
TSFSE (Transmit Stereo Frame Sync Enable)
0 - Normal mode
1 - Frame sync becomes L/R clock
TXSE (TxSEC Enable)
0 - Secondary side disabled
1 - Secondary side enabled

Figure 24-26. SPORT Transmit Configuration 2 Register

Additional information for the SPORT_TCR1 and SPORT_TCR2 transmit configuration register bits includes:

- **Transmit enable** (TSPEN). This bit selects whether the SPORT is enabled to transmit (if set) or disabled (if cleared).

  Setting TSPEN causes an immediate assertion of a SPORT TX interrupt, indicating that the TX data register is empty and needs to be filled. This is normally desirable because it allows centralization of the transmit data write code in the TX interrupt service routine (ISR). For this reason, the code should initialize the ISR and be ready to service TX interrupts before setting TSPEN.

  Similarly, if DMA transfers are used, DMA control should be configured correctly before setting TSPEN. Set all DMA control registers before setting TSPEN.

Clearing TSPEN causes the SPORT to stop driving data, TSCLK, and frame sync pins; it also shuts down the internal SPORT circuitry. In low power applications, battery life can be extended by clearing TSPEN whenever the SPORT is not in use.

All SPORT control registers should be programmed before TSPEN is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write SPORT_TCR1 with all of the necessary bits, including TSPEN.

- **Internal transmit clock select**. (ITCLK). This bit selects the internal transmit clock (if set) or the external transmit clock on the TSCLK pin (if cleared). The TCLKDIV MMR value is not used when an external clock is selected.
- **Data formatting type select**. The two TDTYPE bits specify data formats used for single and multichannel operation.
- **Bit order select**. (TLSBIT). The TLSBIT bit selects the bit order of the data words transmitted over the SPORT.
- **Serial word length select**. (SLEN). The serial word length (the number of bits in each word transmitted over the SPORTs) is calculated by adding 1 to the value of the SLEN field:

  Serial Word Length = SLEN + 1;

  The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field. Three common settings for the SLEN field are 15, to transmit a full 16-bit word; 7, to transmit an 8-bit byte; and 23, to transmit a 24-bit word. The processor can load 16- or 32-bit values into the transmit buffer via DMA or an MMR write

instruction; the SLEN field tells the SPORT how many of those bits to shift out of the register over the serial link. The SPORT always transfers the SLEN+1 lower bits from the transmit buffer.

The frame sync signal is controlled by the SPORT_TFSDIV and SPORT_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync divider must be programmed into the frame sync divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal transmit frame sync select**. (ITFS). This bit selects whether the SPORT uses an internal TFS (if set) or an external TFS (if cleared).
- **Transmit frame sync required select**. (TFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a transmit frame sync for every data word.

The TFSR bit is normally set during SPORT configuration. A frame sync pulse is used to mark the beginning of each word or data packet, and most systems need a frame sync to function properly.

- **Data-Independent transmit frame sync select**. (DITFS). This bit selects whether the SPORT generates a data-independent TFS (sync at selected interval) or a data-dependent TFS (sync when data is present in SPORT_TX) for the case of internal frame sync select (ITFS = 1). The DITFS bit is ignored when external frame syncs are selected.

  The frame sync pulse marks the beginning of the data word. If DITFS is set, the frame sync pulse is issued on time, whether the SPORT_TX register has been loaded or not; if DITFS is cleared, the frame sync pulse is only generated if the SPORT_TX data register has been loaded. If the receiver demands regular frame sync pulses, DITFS should be set, and the processor should keep loading the

SPORT_TX register on time. If the receiver can tolerate occasional late frame sync pulses, DITFS should be cleared to prevent the SPORT from transmitting old data twice or transmitting garbled data if the processor is late in loading the SPORT_TX register.

- **Low transmit frame sync select**. (LTFS). This bit selects an active low TFS (if set) or active high TFS (if cleared).
- **Late transmit frame sync**. (LATFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
- **Clock drive/sample edge select**. (TCKFE). This bit selects which edge of the TCLKx signal the SPORT uses for driving data, for driving internally generated frame syncs, and for sampling externally generated frame syncs. If set, data and internally generated frame syncs are driven on the falling edge, and externally generated frame syncs are sampled on the rising edge. If cleared, data and internally generated frame syncs are driven on the rising edge, and externally generated frame syncs are sampled on the falling edge.
- **TxSec enable**. (TXSE). This bit enables the transmit secondary side of the SPORT (if set).
- **Stereo serial enable**. (TSFSE). This bit enables the stereo serial operating mode of the SPORT (if set). By default this bit is cleared, enabling normal clocking and frame sync.
- **Left/Right order**. (TRFST). If this bit is set, the right channel is transmitted first in stereo serial operating mode. By default this bit is cleared, and the left channel is transmitted first.

SPORT Receive Configuration (SPORT_RCR1 and SPORT_RCR2) Registers

The main control registers for the receive portion of each SPORT are the receive configuration registers, SPORT_RCR1 and SPORT_RCR2, shown in Figure 24-27 on page 24-55 and Figure 24-28 on page 24-56.

A SPORT is enabled for receive if bit 0 (RSPEN) of the receive configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT reception.

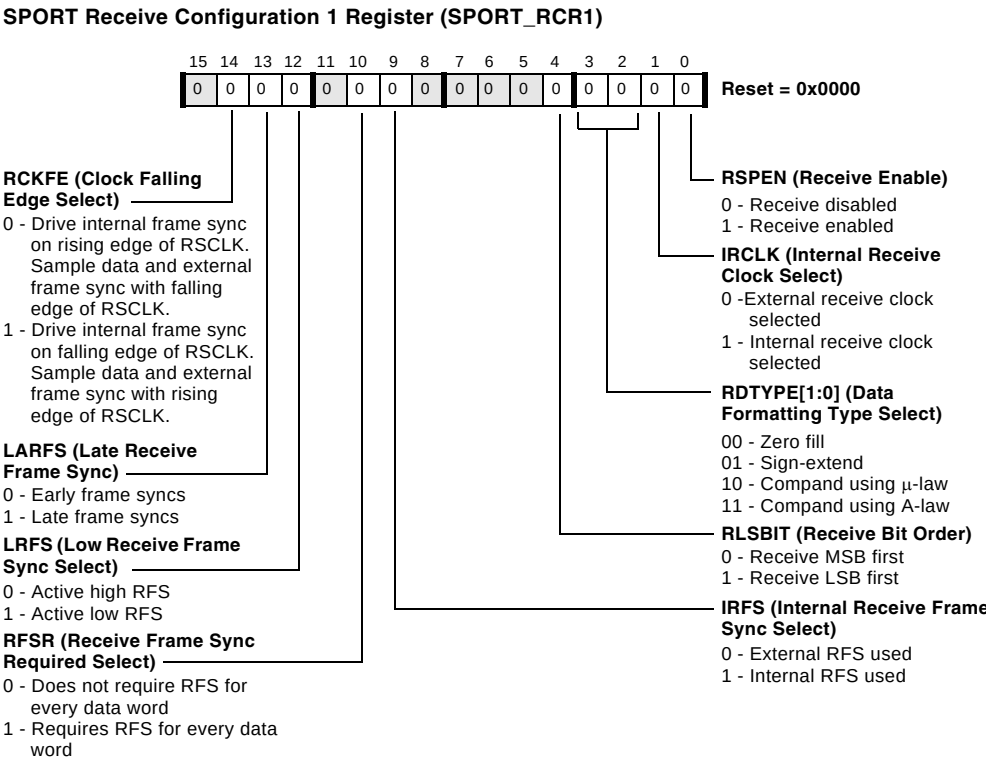

SPORT Receive Configuration 1 Register (SPORT_RCR1)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RCKFE (Clock Falling Edge Select)
0 - Drive internal frame sync on rising edge of RSCLK. Sample data and external frame sync with falling edge of RSCLK.
1 - Drive internal frame sync on falling edge of RSCLK. Sample data and external frame sync with rising edge of RSCLK.
LARFS (Late Receive Frame Sync)
0 - Early frame syncs
1 - Late frame syncs
LRFS (Low Receive Frame Sync Select)
0 - Active high RFS
1 - Active low RFS
RFSR (Receive Frame Sync Required Select)
0 - Does not require RFS for every data word
1 - Requires RFS for every data word
RSPEN (Receive Enable)
0 - Receive disabled
1 - Receive enabled
IRCLK (Internal Receive Clock Select)
0 -External receive clock selected
1 - Internal receive clock selected
RDTYPE[1:0] (Data Formatting Type Select)
00 - Zero fill
01 - Sign-extend
10 - Compand using μ-law
11 - Compand using A-law
RLSBIT (Receive Bit Order)
0 - Receive MSB first
1 - Receive LSB first
IRFS (Internal Receive Frame Sync Select)
0 - External RFS used
1 - Internal RFS used

Figure 24-27. SPORT Receive Configuration 1 Register

When the SPORT is enabled to receive (RSPEN set), corresponding SPORT configuration register writes are not allowed except for SPORT_RCLKDIV and multichannel mode channel select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORT_RCR1 is not written except for bit 0 (RSPEN). For example,

```
write (SPORT_RCR1, 0x0001) ;  /* SPORT RX Enabled */
write (SPORT_RCR1, 0xFF01) ;  /* ignored, no effect */
write (SPORT_RCR1, 0xFFF0) ;  /* SPORT disabled, SPORT_RCR1
                                 still equal to 0x0000 */
```

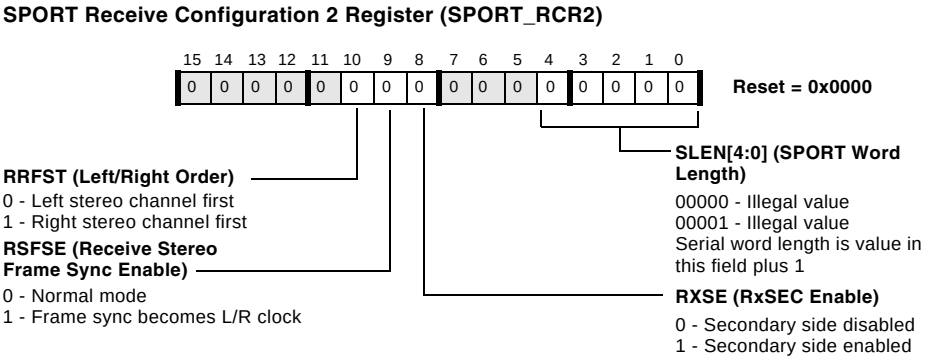

SPORT Receive Configuration 2 Register (SPORT_RCR2)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SLEN[4:0] (SPORT Word Length)
00000 - Illegal value
00001 - Illegal value
Serial word length is value in this field plus 1
RRFST (Left/Right Order)
0 - Left stereo channel first
1 - Right stereo channel first
RSFSE (Receive Stereo Frame Sync Enable)
0 - Normal mode
1 - Frame sync becomes L/R clock
RXSE (RxSEC Enable)
0 - Secondary side disabled
1 - Secondary side enabled

Figure 24-28. SPORT Receive Configuration 2 Register

Additional information for the SPORT_RCR1 and SPORTRCR2 receive configuration register bits:

- **Receive enable**. (RSPEN). This bit selects whether the SPORT is enabled to receive (if set) or disabled (if cleared). Setting the RSPEN bit turns on the SPORT and causes it to sample data from the data receive pins as well as the receive bit clock and receive frame sync pins if so programmed.

  Setting RSPEN enables the SPORT receiver, which can generate a SPORT RX interrupt. For this reason, the code should initialize the ISR and the DMA control registers, and should be ready to service RX interrupts before setting RSPEN. Setting RSPEN also generates DMA requests if DMA is enabled and data is received. Set all DMA control registers before setting RSPEN.

  Clearing RSPEN causes the SPORT to stop receiving data; it also shuts down the internal SPORT receive circuitry. In low power applications, battery life can be extended by clearing RSPEN whenever the SPORT is not in use.

All SPORT control registers should be programmed before RSPEN is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write SPORT_RCR1 with all of the necessary bits, including RSPEN.

- **Internal receive clock select**. (IRCLK). This bit selects the internal receive clock (if set) or external receive clock (if cleared). The RCLKDIV MMR value is not used when an external clock is selected.
- **Data formatting type select**. (RDTYPE). The two RDTYPE bits specify one of four data formats used for single and multichannel operation.
- **Bit order select**. (RLSBIT). The RLSBIT bit selects the bit order of the data words received over the SPORTs.
- **Serial word length select**. (SLEN). The serial word length (the number of bits in each word received over the SPORTs) is calculated by adding 1 to the value of the SLEN field. The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field.

The frame sync signal is controlled by the SPORT_TFSDIV and SPORT_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync divider must be programmed into the frame sync divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal receive frame sync select**. (IRFS). This bit selects whether the SPORT uses an internal RFS (if set) or an external RFS (if cleared).
- **Receive frame sync required select**. (RFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a receive frame sync for every data word.

- **Low receive frame sync select**. (LRFS). This bit selects an active low RFS (if set) or active high RFS (if cleared).
- **Late receive frame sync**. (LARFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
- **Clock drive/sample edge select**. (RCKFE). This bit selects which edge of the RSCLK clock signal the SPORT uses for sampling data, for sampling externally generated frame syncs, and for driving internally generated frame syncs. If set, internally generated frame syncs are driven on the falling edge, and data and externally generated frame syncs are sampled on the rising edge. If cleared, internally generated frame syncs are driven on the rising edge, and data and externally generated frame syncs are sampled on the falling edge.
- **RxSec enable**. (RXSE). This bit enables the receive secondary side of the SPORT (if set).
- **Stereo serial enable**. (RSFSE). This bit enables the stereo serial operating mode of the SPORT (if set). By default this bit is cleared, enabling normal clocking and frame sync.
- **Left/Right order**. (RRFST). If this bit is set, the right channel is received first in stereo serial operating mode. By default this bit is cleared, and the left channel is received first.

Data Word Formats

The format of the data words transferred over the SPORTs is configured by the combination of transmit SLEN and receive SLEN; RDTYPE; TDTYPE; RLSBIT; and TLSBIT bits of the SPORT_TCR1, SPORT_TCR2, SPORT_RCR1, and SPORT_RCR2 registers.

SPORT Transmit Data (SPORT_TX) Register

The SPORT_TX register is a write-only register. Reads produce a peripheral bus error. Writes to this register cause writes into the transmitter FIFO. The 16-bit wide FIFO is 8 deep for word length ≤ 16 and 4 deep for word length > 16. The FIFO is common to both primary and secondary data and stores data for both. Data ordering in the FIFO is shown in the Figure 24-29. The SPORT_TX register is shown in Figure 24-30 on page 24-61.

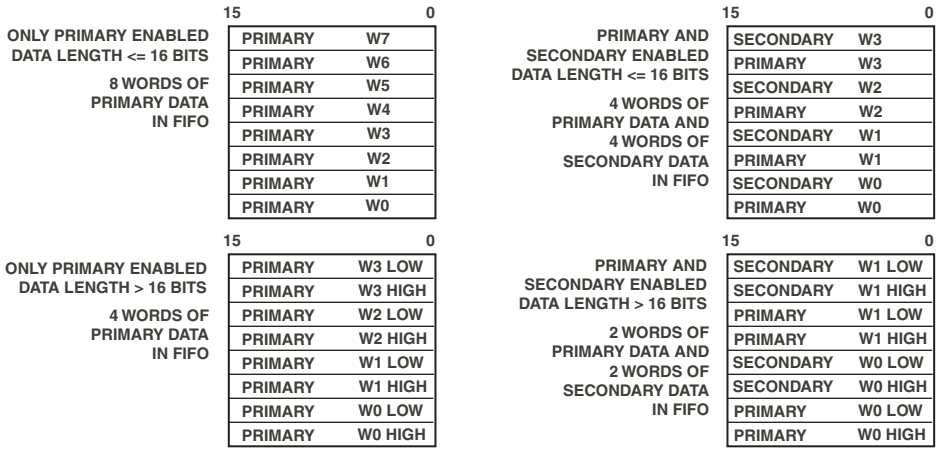


Figure 24-29. SPORT Transmit FIFO Data Ordering

It is important to keep the interleaving of primary and secondary data in the FIFO as shown. This means that peripheral bus/DMA writes to the FIFO must follow an order of primary first, and then secondary, if secondary is enabled. DAB/peripheral bus writes must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit write. Use a 32-bit write for word length greater than 16 bits.

When transmit is enabled, data from the FIFO is assembled in the TX Hold register based on TXSE and SLEN, and then shifted into the primary

and secondary shift registers. From here, the data is shifted out serially on the DTPRI and DTSEC pins.

The SPORT TX interrupt is asserted when TSPEN = 1 and the TX FIFO has room for additional words. This interrupt does not occur if SPORT DMA is enabled.

The transmit underflow status bit (TUVF) is set in the SPORT_STAT register when a transmit frame sync occurs and no new data has been loaded into the serial shift register. In multichannel mode (MCM), TUVF is set whenever the serial shift register is not loaded, and transmission begins on the current enabled channel. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the SPORT (writing TXEN = 0).

If software causes the core processor to attempt a write to a full TX FIFO with a SPORT_TX write, the new data is lost and no overwrites occur to data in the FIFO. The TOVF status bit is set and a SPORT error interrupt is asserted. The TOVF bit is a sticky bit; it is only cleared by disabling the SPORT TX. To find out whether the core processor can access the SPORT_TX register without causing this type of error, read the register's status first. The TXF bit in the SPORT_STAT register is 0 if space is available for another word in the FIFO.

The TXF and TOVF status bits in the SPORT_STAT register are updated upon writes from the core processor, even when the SPORT is disabled.

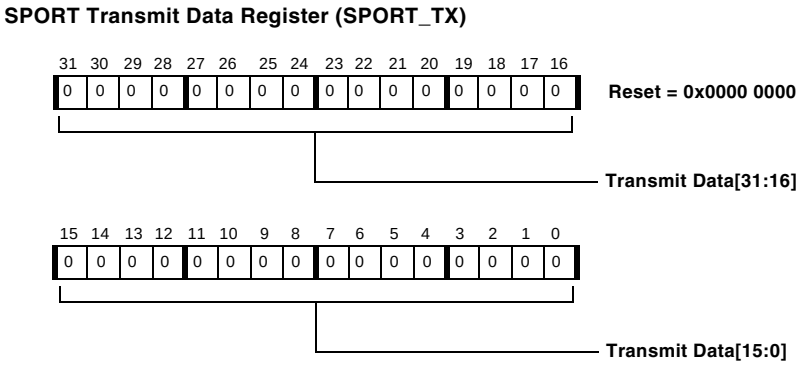


Figure 24-30. SPORT Transmit Data Register

SPORT Receive Data (SPORT_RX) Register

The SPORT_RX register is a read-only register. Writes produce a peripheral bus error. The same location is read for both primary and secondary data. Reading from this register space causes reading of the receive FIFO. This 16-bit FIFO is 8 deep for receive word length ≤ 16 and 4 deep for length > 16 bits. The FIFO is shared by both primary and secondary receive data. The order for reading using peripheral bus/DMA reads is important since data is stored in differently depending on the setting of the SLEN and RXSE configuration bits.

Data storage and data ordering in the FIFO are shown in Figure 24-31 on page 24-62. The SPORT_RX register is shown in Figure 24-32 on page 24-63.

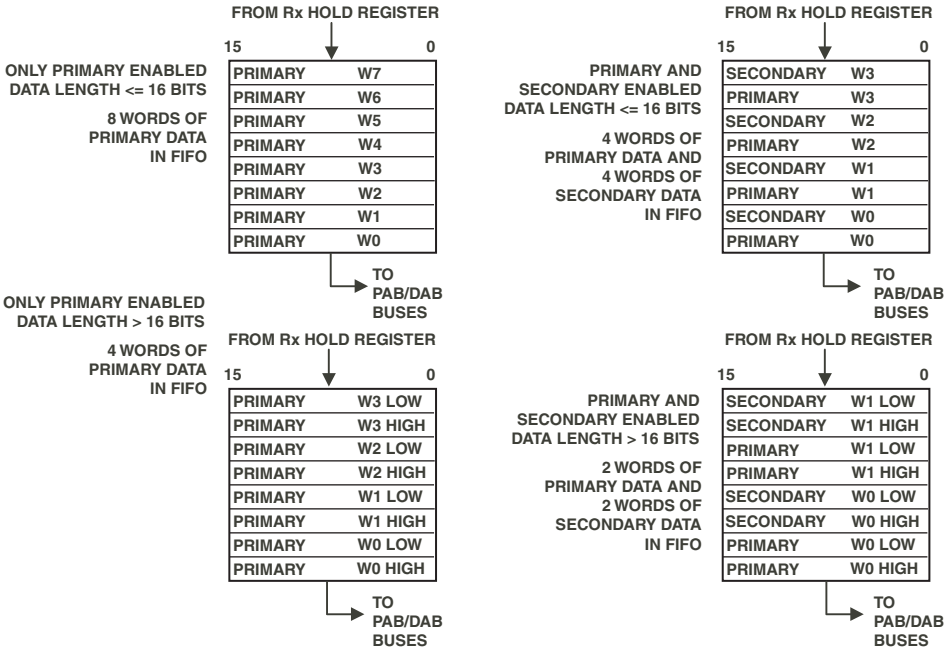


Figure 24-31. SPORT Receive FIFO Data Ordering

When reading from the FIFO for both primary and secondary data, read primary first, followed by secondary. DAB/peripheral bus reads must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit read. Use a 32-bit read for word length greater than 16 bits.

When receiving is enabled, data from the DRPRI pin is loaded into the RX primary shift register, while data from the DRSEC pin is loaded into the RX secondary shift register. At transfer completion of a word, data is shifted into the RX hold registers for primary and secondary data, respectively. Data from the hold registers is moved into the FIFO based on RXSE and SLEN.

The SPORT RX interrupt is generated when RSPEN = 1 and the RX FIFO has received words in it. When the core processor has read all the words in the FIFO, the RX interrupt is cleared. The SPORT RX interrupt is set only if SPORT RX DMA is disabled; otherwise, the FIFO is read by DMA reads.

If the program causes the core processor to attempt a read from an empty RX FIFO, old data is read, the RUVF flag is set in the SPORT_STAT register, and the SPORT error interrupt is asserted. The RUVF bit is a sticky bit and is cleared only when the SPORT is disabled. To determine if the core can access the RX registers without causing this error, first read the RX FIFO status (RXNE in the SPORT_STAT register). The RUVF status bit is updated even when the SPORT is disabled.

The ROVF status bit is set in the SPORT_STAT register when a new word is assembled in the RX shift register and the RX hold register has not moved the data to the FIFO. The previously written word in the hold register is overwritten. The ROVF bit is a sticky bit; it is only cleared by disabling the SPORT RX.

SPORT Receive Data Register (SPORT_RX)

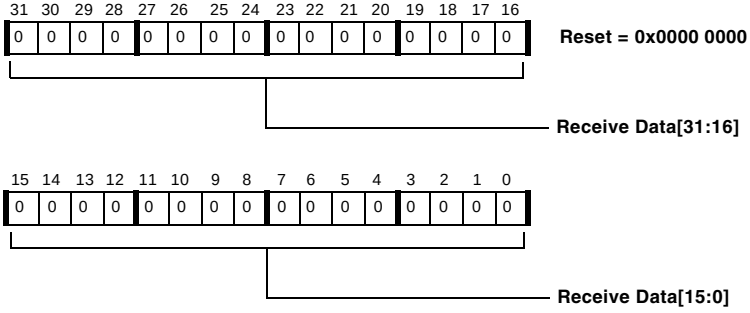


Figure 24-32. SPORT Receive Data Register

SPORT Status (SPORT_STAT) Register

The SPORT_STAT register is used to determine if the access to a SPORT RX or TX FIFO can be made by determining their full or empty status. This register is shown in Figure 24-33 on page 24-65.

The TXF bit in the SPORT_STAT register indicates whether there is room in the TX FIFO. The RXNE status bit indicates whether there are words in the RX FIFO. The TXHRE bit indicates if the TX hold register is empty.

The transmit underflow status bit (TUVF) is set whenever the TFS signal occurs (from either an external or internal source) while the TX shift register is empty. The internally generated TFS may be suppressed whenever SPORT_TX is empty by clearing the DITFS control bit in the SPORT_TCR1 register. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the SPORT (writing TXEN = 0).

For continuous transmission (TFSR = 0), TUVF is set at the end of a transmitted word if no new word is available in the TX hold register.

The TOVF bit is set when a word is written to the TX FIFO when it is full. It is a sticky W1C bit and is also cleared by writing TXEN = 0. Both TXF and TOVF are updated even when the SPORT is disabled.

When the SPORT RX hold register is full, and a new receive word is received in the shift register, the receive overflow status bit (ROVF) is set in the SPORT_STAT register. It is a sticky W1C bit and is also cleared by disabling the SPORT (writing RXEN = 0).

The RUVF bit is set when a read is attempted from the RX FIFO and it is empty. It is a sticky W1C bit and is also cleared by writing RXEN = 0. The RUVF bit is updated even when the SPORT is disabled.

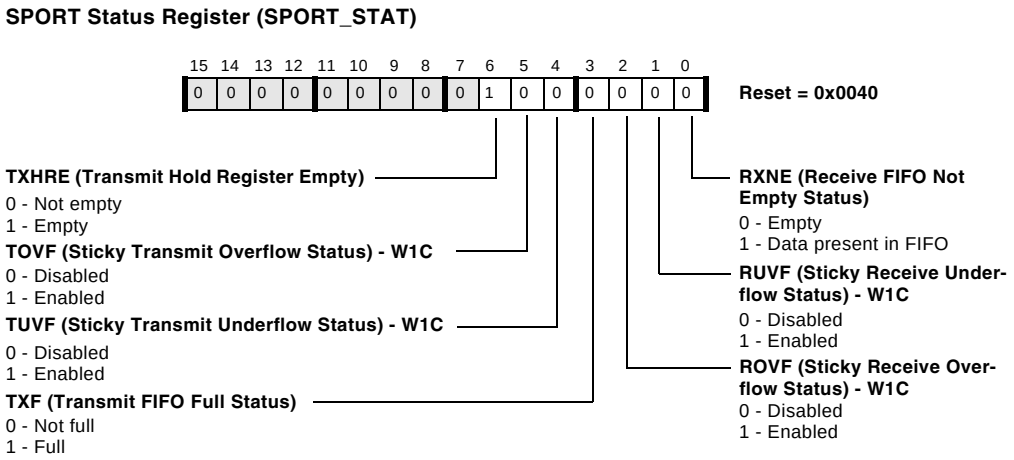

SPORT Status Register (SPORT_STAT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
Reset = 0x0040
TXHRE (Transmit Hold Register Empty)
0 - Not empty
1 - Empty
TOVF (Sticky Transmit Overflow Status) - W1C
0 - Disabled
1 - Enabled
TUVF (Sticky Transmit Underflow Status) - W1C
0 - Disabled
1 - Enabled
TXF (Transmit FIFO Full Status)
0 - Not full
1 - Full
RXNE (Receive FIFO Not Empty Status)
0 - Empty
1 - Data present in FIFO
RUVF (Sticky Receive Underflow Status) - W1C
0 - Disabled
1 - Enabled
ROVF (Sticky Receive Overflow Status) - W1C
0 - Disabled
1 - Enabled

Figure 24-33. SPORT Status Register

SPORT Transmit and Receive Serial Clock Divider (SPORT_TCLKDIV and SPORT_RCLKDIV) Registers

The frequency of an internally generated clock is a function of the system clock frequency (as seen at the SCLK pin) and the value of the 16-bit serial clock divide modulus registers (the SPORT_TCLKDIV register, shown in Figure 24-34, and the SPORT_RCLKDIV register, shown in Figure 24-35 on page 24-66).

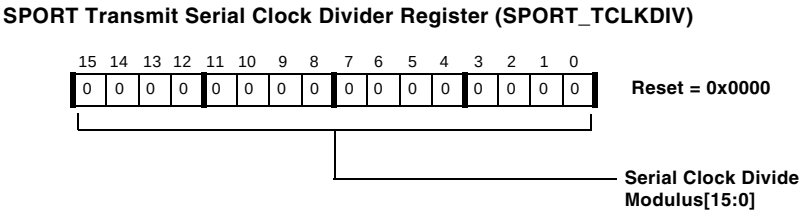

SPORT Transmit Serial Clock Divider Register (SPORT_TCLKDIV)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Serial Clock Divide Modulus[15:0]

Figure 24-34. SPORT Transmit Serial Clock Divider Register

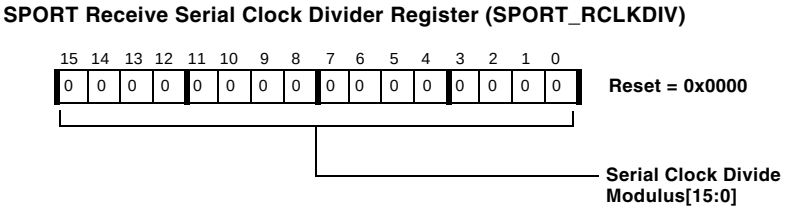

SPORT Receive Serial Clock Divider Register (SPORT_RCLKDIV)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Serial Clock Divide
Modulus[15:0]

Figure 24-35. SPORT Receive Serial Clock Divider Register

SPORT Transmit and Receive Frame Sync Divider (SPORT_TFSDIV and SPORT_RFSDIV) Registers

The 16-bit `SPORT_TFSDIV` and `SPORT_RFSDIV` registers specify how many transmit or receive clock cycles are counted before generating a `TFS` or `RFS` pulse when the frame sync is internally generated. In this way, a frame sync can be used to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks. These registers are shown in Figure 24-36 and Figure 24-37 on page 24-67.

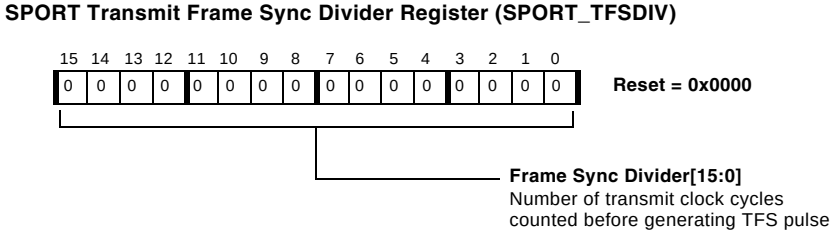

SPORT Transmit Frame Sync Divider Register (SPORT_TFSDIV)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Frame Sync Divider[15:0]
Number of transmit clock cycles
counted before generating TFS pulse

Figure 24-36. SPORT Transmit Frame Sync Divider Register

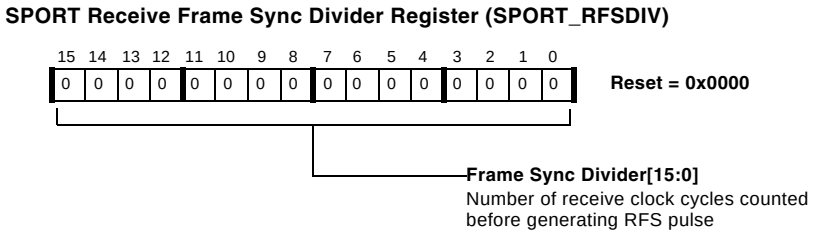

SPORT Receive Frame Sync Divider Register (SPORT_RFSDIV)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Frame Sync Divider[15:0]
Number of receive clock cycles counted
before generating RFS pulse

Figure 24-37. SPORT Receive Frame Sync Divider Register

SPORT Multichannel Configuration (SPORT_MCMC1 and SPORT_MCMC2) Registers

There are two multichannel configuration registers for each SPORT, shown in Figure 24-38 and Figure 24-39 on page 24-68. These registers are used to configure the multichannel operation of the SPORT. The two control registers are shown below.

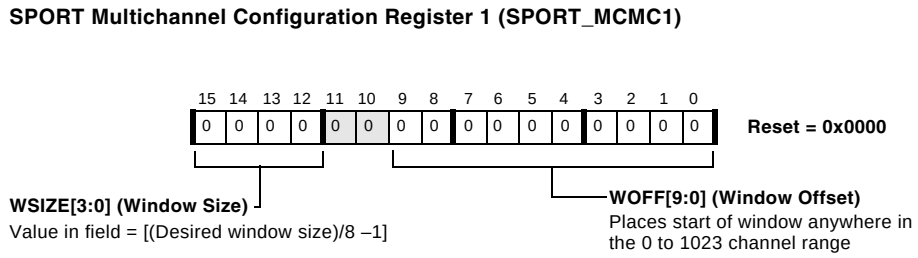

SPORT Multichannel Configuration Register 1 (SPORT_MCMC1)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
WSIZE[3:0] (Window Size)
Value in field = [(Desired window size)/8 –1]
WOFF[9:0] (Window Offset)
Places start of window anywhere in
the 0 to 1023 channel range

Figure 24-38. SPORT Multichannel Configuration Register 1

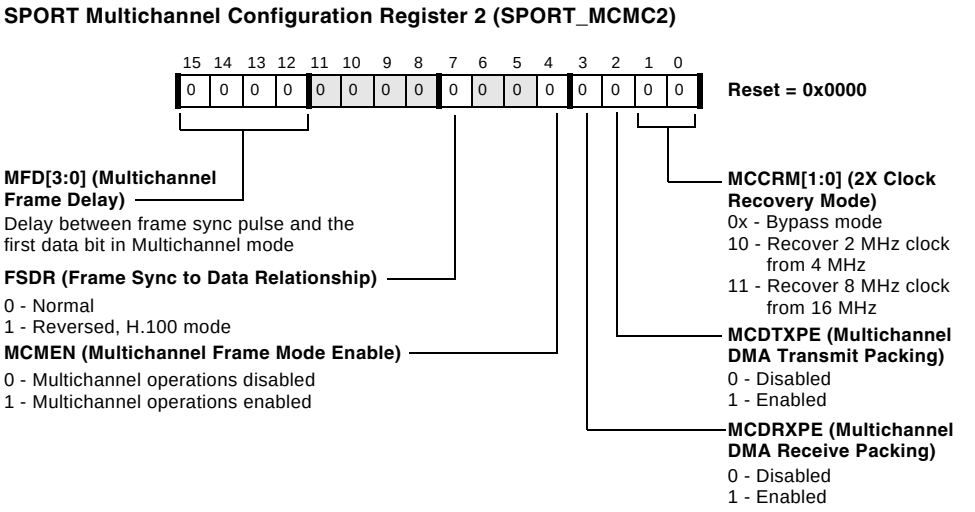

SPORT Multichannel Configuration Register 2 (SPORT_MCMC2)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
MFD[3:0] (Multichannel Frame Delay)
Delay between frame sync pulse and the first data bit in Multichannel mode
FSDR (Frame Sync to Data Relationship)
0 - Normal
1 - Reversed, H.100 mode
MCMEN (Multichannel Frame Mode Enable)
0 - Multichannel operations disabled
1 - Multichannel operations enabled
MCCRM[1:0] (2X Clock Recovery Mode)
0x - Bypass mode
10 - Recover 2 MHz clock from 4 MHz
11 - Recover 8 MHz clock from 16 MHz
MCDTXPE (Multichannel DMA Transmit Packing)
0 - Disabled
1 - Enabled
MCDRXPE (Multichannel DMA Receive Packing)
0 - Disabled
1 - Enabled

Figure 24-39. SPORT Multichannel Configuration Register 2

SPORT Current Channel (SPORT_CHNL) Register

The 10-bit `CHNL` field in the `SPORT_CHNL` register indicates which channel is currently being serviced during multichannel operation. This field is a read-only status indicator. The `CHNL[9:0]` field increments by one as each channel is serviced. The counter stops at the upper end of the defined window. The channel select register restarts at 0 at each frame sync. As an example, for a window size of 8 and an offset of 148, the counter displays a value between 0 and 156.

Once the window size has completed, the channel counter resets to 0 in preparation for the next frame. Because there are synchronization delays between `RSCLK` and the processor clock, the channel register value is approximate. It is never ahead of the channel being served, but it may lag behind. See Figure 24-40 on page 24-69.

SPORT Current Channel Register (SPORT_CHNL)
RO

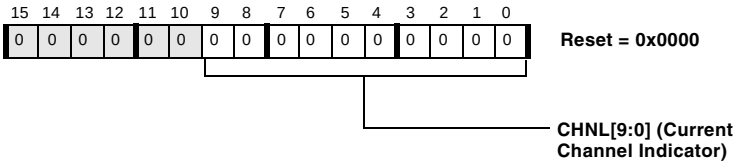


Figure 24-40. SPORT Current Channel Register

SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers

The SPORT_MRCSn registers (shown in Figure 24-41 on page 24-70) are used to enable and disable individual channels. They specify the active receive channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the SPORT selects that word for receive from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in the SPORT_MRCSn register causes the SPORT to receive the word in that channel's position of the datastream; the received word is loaded into the RX buffer. When the secondary receive side is enabled by the RXSE bit, both inputs are processed on enabled channels. Clearing the bit in the SPORT_MRCSn register causes the SPORT to ignore the data on either channel.

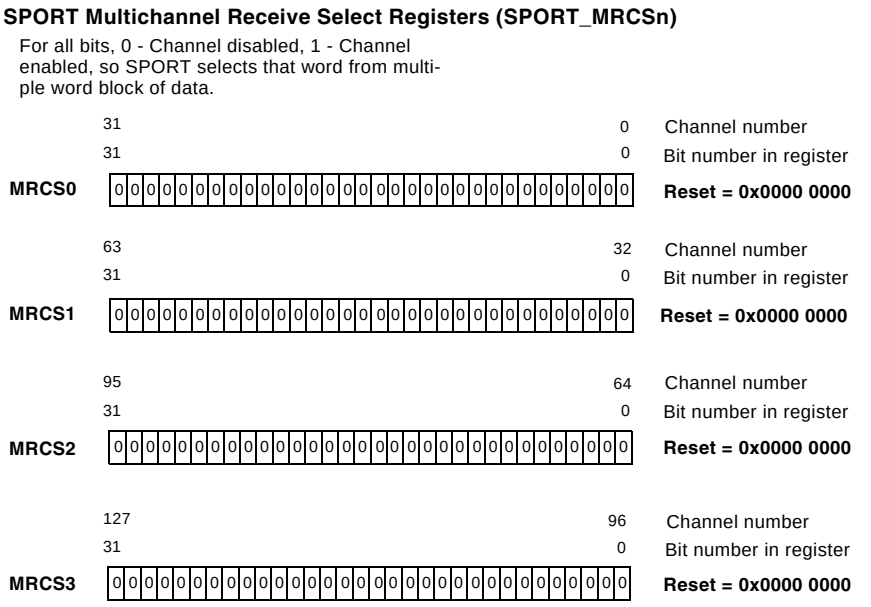

SPORT Multichannel Receive Select Registers (SPORT_MRCSn)
For all bits, 0 - Channel disabled, 1 - Channel enabled, so SPORT selects that word from multiple word block of data.
31
0
Channel number
31
0
Bit number in register
MRCS0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
63
32
Channel number
31
0
Bit number in register
MRCS1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
95
64
Channel number
31
0
Bit number in register
MRCS2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
127
96
Channel number
31
0
Bit number in register
MRCS3
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000

Figure 24-41. SPORT Multichannel Receive Select Registers

SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers

The SPORT_MTCSn registers (shown in Figure 24-42 on page 24-71) are used to enable and disable individual channels. They specify the active transmit channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the SPORT selects that word for transmit from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in a SPORT_MTCSn register causes the SPORT to transmit the word in that channel's position of the datastream. When the secondary transmit side is enabled by the TXSE bit, both sides transmit a

word on the enabled channel. Clearing the bit in the SPORT_MTCSn register causes a SPORT controllers' data transmit pins to three-state during the time slot of that channel.

SPORT Multichannel Transmit Select Registers (SPORT_MTCSn)

For all bits, 0 - Channel disabled, 1 - Channel enabled, so SPORT selects that word from multiple word block of data.

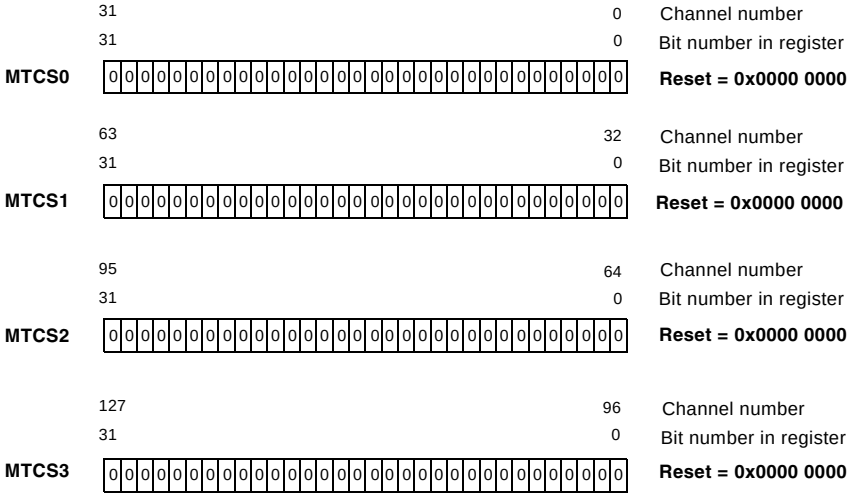


Figure 24-42. SPORT Multichannel Transmit Select Registers

Programming Examples

This section shows an example of typical usage of the SPORT peripheral in conjunction with the DMA controller. See Listing 24-1 on page 24-72 through Listing 24-4 on page 24-77. These listings assume a processor with at least two SPORTs, SPORT0 and SPORT1.

The SPORT is usually employed for high-speed, continuous serial transfers. The example reflects this, in that the SPORT is set-up for auto-buffered, repeated DMA transfers.

Because of the many possible configurations, the example uses generic labels for the content of the SPORT's configuration registers (SPORT_RCRn and SPORT_TCRn) and the DMA configuration. An example value is given in the comments, but for the meaning of the individual bits the user is referred to the detailed explanation in this chapter.

The example configures both the receive and the transmit section. Since they are completely independent, the code uses separate labels.

SPORT Initialization Sequence

The SPORT's receiver and transmitter are configured, but they are not enabled yet.

Listing 24-1. SPORT Initialization

```
Program_SPORT_TRANSMITTER_Registers:
  /* Set P0 to SPORT0 Base Address */
  P0.h = hi(SPORT0_TCR1);
  P0.l = lo(SPORT0_TCR1);

/* Configure Clock speeds */
  R1 = SPORT_TCLK_CONFIG;  /* Divider SCLK/TCLK (value 0 to
65535) */
  W[P0 + (SPORT0_TCLKDIV - SPORT0_TCR1)] = R1;
                        /* TCK divider register */
  /* number of Bitclocks between FrameSyncs -1  (value SPORT_SLEN
to 65535) */
  R1 = SPORT_TFSDIV_CONFIG;
  W[P0 + (SPORT0_TFSDIV - SPORT0_TCR1)] = R1;
                       /* TFSDIV register */

/* Transmit configuration */
```

```
  /* Configuration register 2   (for instance 0x000E for 16-bit
wordlength) */
  R1 = SPORT_TRANSMIT_CONF_2;
  W[P0 + (SPORT0_TCR2 - SPORT0_TCR1)] = R1;
  /* Configuration register 1   (for instance 0x4E12 for inter-
nally generated clk and framesync) */
  R1 = SPORT_TRANSMIT_CONF_1;
  W[P0] = R1;
  ssync;
/* NOTE: SPORT0 TX NOT enabled yet (bit 0 of TCR1 must be zero) */

Program_SPORT_RECEIVER_Registers:
  /* Set P0 to SPORT0 Base Address */
  P0.h = hi(SPORT0_RCR1);
  P0.l = lo(SPORT0_RCR1);

/* Configure Clock speeds */
  R1 = SPORT_RCLK_CONFIG;  /* Divider SCLK/RCLK (value 0 to
65535) */
  W[P0 + (SPORT0_RCLKDIV - SPORT0_RCR1)] = R1;   /* RCK divider
register */
  /* number of Bitclock between FrameSyncs -1  (value SPORT_SLEN
to 65535) */
  R1 = SPORT_RFSDIV_CONFIG;
  W[P0 + (SPORT0_RFSDIV - SPORT0_RCR1)] = R1;
                               /* RFSDIV register */

/* Receive configuration */
  /* Configuration register 2   (for instance 0x000E for 16-bit
wordlength) */
  R1 = SPORT_RECEIVE_CONF_2;
  W[P0 + (SPORT0_RCR2 - SPORT0_RCR1)] = R1;
  /* Configuration register 1   (for instance 0x4410 for external
clk and framesync) */
```

```
  R1 = SPORT_RECEIVE_CONF_1;
  W[P0] = R1;
  ssync;  /* NOTE: SPORT0 RX NOT enabled yet (bit 0 of RCR1 must
be zero) */
```

DMA Initialization Sequence

Next the DMA channels for receive (channel3 in this example) and for transmit (channel4 in this example) are set up for auto-buffered, one-dimensional, 32-bit transfers. Again, there are other possibilities, so generic labels have been used, with a particular value shown in the comments.

Note that the DMA channels can be enabled at the end of the configuration since the SPORT is not enabled yet. However, if preferred, the user can enable the DMA later, immediately before enabling the SPORT. The only requirement is that the DMA channel be enabled before the associated peripheral is enabled to start the transfer.

Listing 24-2. DMA Initialization

```
Program_DMA_Controller:

/* Receiver (DMA channel 3) */
  /* Set P0 to DMA Base Address */
  P0.l = lo(DMA3_CONFIG);
  P0.h = hi(DMA3_CONFIG);

  /* Configuration (for instance 0x108A for Autobuffer, 32-bit
wide transfers) */
  R0 = DMA_RECEIVE_CONF(z);
  W[P0] = R0;  /* configuration register */
```

```
  /* rx_buf = Buffer in Data memory (divide count by four because
of 32-bit DMA transfers) */
  R1 = (length(rx_buf)/4)(z);
  W[P0 + (DMA3_X_COUNT - DMA3_CONFIG)] = R1;
                                        /* X_count register */
  R1 = 4(z);  /* 4 bytes in a 32-bit transfer */
  W[P0 + (DMA3_X_MODIFY - DMA3_CONFIG)] = R1;
                                       /* X_modify register */

            /* start_address register points to memory buffer
                to be filled */
  R1.l = rx_buf;
  R1.h = rx_buf;
  [P0 + (DMA3_START_ADDR - DMA3_CONFIG)] = R1;

  BITSET(R0,0);  /* R0 still contains value of CONFIG register -
set bit 0 */
  W[P0] = R0;    /* enable DMA channel (SPORT not enabled yet) */

/* Transmitter (DMA channel 4) */
  /* Set P0 to DMA Base Address */
  P0.l = lo(DMA4_CONFIG);
  P0.h = hi(DMA4_CONFIG);
  /* Configuration (for instance 0x1088 for Autobuffer, 32-bit
wide transfers) */
  R0 = DMA_TRANSMIT_CONF(z);
  W[P0] = R0;  /* configuration register */

  /* tx_buf = Buffer in Data memory (divide count by four because
of 32-bit DMA transfers) */
  R1 = (length(tx_buf)/4)(z);
  W[P0 + (DMA4_X_COUNT - DMA4_CONFIG)] = R1;
              /* X_count register */
  R1 = 4(z);  /* 4 bytes in a 32-bit transfer */
```

```
W[P0 + (DMA4_X_MODIFY - DMA4_CONFIG)] = R1;
            /* X_modify register */

/* start_address register points to memory buffer to be
   transmitted from */
R1.l = tx_buf;
R1.h = tx_buf;
[P0 + (DMA4_START_ADDR - DMA4_CONFIG)] = R1;

BITSET(R0,0);  /* R0 still contains value of CONFIG register -
                  set bit 0 */
W[P0] = R0;    /* enable DMA channel (SPORT not enabled yet) */
```

Interrupt Servicing

The receive channel and the transmit channel will each generate an interrupt request if so programmed. The following code fragments show the minimum actions that must be taken. Not shown is the programming of the core and system event controllers.

Listing 24-3. Servicing an Interrupt

```
RECEIVE_ISR:
  [--SP] = RETI;  /* nesting of interrupts */

  /* clear DMA interrupt request */
  P0.h = hi(DMA3_IRQ_STATUS);
  P0.l = lo(DMA3_IRQ_STATUS);
  R1   = 1;
  W[P0] = R1.l;  /* write one to clear */

  RETI = [SP++];
  rti;
```

```
TRANSMIT_ISR:
  [--SP] = RETI;  /* nesting of interrupts */

  /* clear DMA interrupt request */
  P0.h = hi(DMA4_IRQ_STATUS);
  P0.l = lo(DMA4_IRQ_STATUS);
  R1   = 1;
  W[P0] = R1.l;  /* write one to clear */

  RETI = [SP++];
  rti;
```

Starting a Transfer

After the initialization procedure outlined in the previous sections, the receiver and transmitter are enabled. The core may just wait for interrupts.

Listing 24-4. Starting a Transfer

```
/* Enable Sport0 RX and TX */
  P0.h = hi(SPORT0_RCR1);
  P0.l = lo(SPORT0_RCR1);
  R1 = W[P0](Z);
  BITSET(R1,0);
  W[P0] = R1;
  ssync;  /* Enable Receiver (set bit 0) */

  P0.h = hi(SPORT0_TCR1);
  P0.l = lo(SPORT0_TCR1);
  R1 = W[P0](Z);
  BITSET(R1,0);
  W[P0] = R1;
  ssync;  /* Enable Transmitter (set bit 0) */
```

```
/* dummy wait loop (do nothing but waiting for interrupts) */
wait_forever:
  jump wait_forever;
```

Unique Information for the ADSP-BF52x Processor

None.

25 UART PORT CONTROLLERS

This chapter describes the universal asynchronous receiver/transmitter (UART) module. Following an overview and a list of key features is a description of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF52x

For details regarding the number of UARTs for the ADSP-BF52x product, please refer to the *ADSP-BF522/523/524/525/526/527 Embedded Processor Data Sheet*.

For UART DMA channel assignments, refer to Table 6-7 on page 6-106 in Chapter 6, “Direct Memory Access”.

For UART interrupt vector assignments, refer to Table 5-3 on page 5-18 in Chapter 5, “System Interrupts”.

To determine how each of the UARTs is multiplexed with other functional pins, refer to Table 9-2 on page 9-5 through Table 9-5 on page 9-9 in Chapter 9, “General-Purpose Ports”.

For a list of MMR addresses for each UART, refer to Appendix A, “System MMR Assignments”.

UART behavior for the ADSP-BF52x that differs from the general information in this chapter can be found at the end of this chapter in the section “Unique Information for the ADSP-BF52x Processor” on page 25-43

Overview

The UART module is a full-duplex peripheral compatible with PC-style industry-standard UARTs, sometimes called serial controller interfaces (SCI). UARTs convert data between serial and parallel formats. The serial communication follows an asynchronous protocol that supports various word length, stop bits, bit rate, and parity generation options.

Features

Each UART includes these features:

- 5 – 8 data bits
- 1 or 2 stop bits (1½ in 5-bit mode)
- Even, odd, and sticky parity bit options
- 3 interrupt outputs for reception, transmission, and status
- Independent DMA operation for receive and transmit
- SIR IrDA operation mode
- Internal loop back

The UART is logically compliant to EIA-232E, EIA-422, EIA-485 and LIN standards, but usually requires an external transceiver device to meet electrical requirements. In IrDA® (Infrared Data Association) mode, the UART meets the half-duplex IrDA SIR (9.6/115.2 Kbps rate) protocol.

Interface Overview

Figure 25-1 on page 25-3 shows a simplified block diagram of a UART module and how it interconnects to the Blackfin architecture and to the outside world.

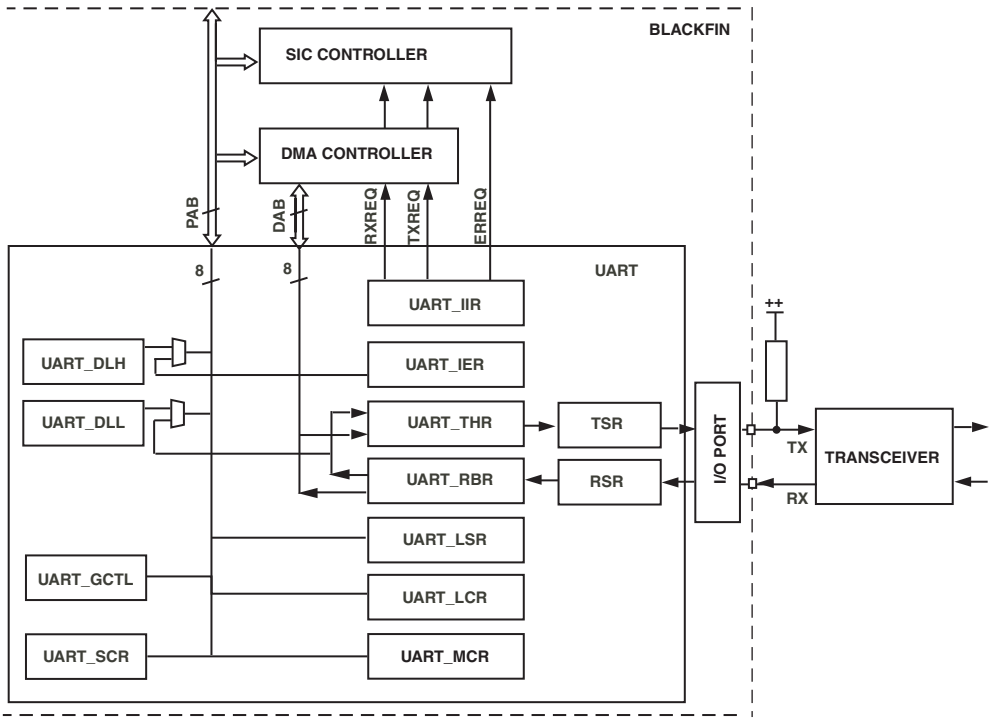


Figure 25-1. UART Block Diagram

External Interface

Each UART features an RX and a TX pin. These two pins usually connect to an external transceiver device that meets the electrical requirements of

full duplex (for example, EIA-232, EIA-422, 4-wire EIA-485) or half duplex (for example, 2-wire EIA-485, LIN) standards.

The RX and TX pins do not need to be used together. If only receive or transmit functionality of a UART module is needed, the unused pin may be used for an alternate function, depending on the port multiplexing scheme of a specific processor. For more details on functionality multiplexed with the UART pins, see Chapter 9, "General-Purpose Ports".

Modem status and control functionality is not supported by the UART modules, but may be implemented using GPIO pins.

Internal Interface

UARTs are DMA-capable peripherals with support for separate TX and RX DMA master channels. They can be used in either DMA or programmed non-DMA mode of operation. The non-DMA mode requires software management of the data flow using either interrupts or polling. The DMA method requires minimal software intervention as the DMA engine itself moves the data. Each UART has its own separate transmit and receive DMA channels. For more information on DMA, see the *Direct Memory Access* chapter.

All UART registers are eight bits wide. They connect to the peripheral bus. However, some registers share their address as controlled by the `DLAB` bit in the `UART_LCR` register. The `UART_RBR` and `UART_THR` registers also connect to the DAB bus

A hardware-assisted autobaud detection mechanism is accomplished by coupling a specific GP Timer with a specific UART. For information on GP Timer - UART pairings for autobaud detection, see General-Purpose Ports chapter.

Description of Operation

The following sections describe the operation of the UART.

UART Transfer Protocol

UART communication follows an asynchronous serial protocol, consisting of individual data words. A word has 5 to 8 data bits.

All data words require a start bit and at least one stop bit. With the optional parity bit, this creates a 7- to 12-bit range for each word. The format of received and transmitted character frames is controlled by the line control register (`UART_LCR`). Data is always transmitted and received least significant bit (LSB) first.

Figure 25-2 on page 25-6 shows a typical physical bitstream measured on one of the TX pins.

Aside from the standard UART functionality, the UART also supports half-duplex serial data communication via infrared signals, according to the recommendations of the Infrared Data Association (IrDA). The physical layer known as IrDA SIR (9.6/115.2 Kbps rate) is based on return-to-zero-inverted (RZI) modulation. Pulse position modulation is not supported.

Using the 16× data rate clock, RZI modulation is achieved by inverting and modulating the non-return-to-zero (NRZ) code normally transmitted by the UART. On the receive side, the 16× clock is used to determine an

IrDA pulse sample window, from which the RZI-modulated NRZ code is recovered.

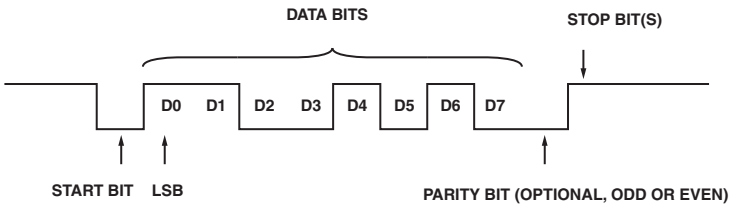


Figure 25-2. Bitstream on a TX Pin Transmitting an “S” Character (0x53)

IrDA support is enabled by setting the IREN bit in the UART_GCTL register. The IrDA application requires external transceivers.

UART Transmit Operation

Receive and transmit paths operate independently except that the bit rate and the frame format are identical for both transfer directions.

Transmission is initiated by writes to the UART_THR register. If no former operation is pending, the data is immediately passed from the UART_THR register to the internal TSR register where it is shifted out at a bit rate equal to SCLK/(16 × Divisor) (see “Bit Rate Generation” on page 25-13 for information about the divisor) with start, stop, and parity bits appended as defined the UART_LCR register. The least significant bit (LSB) is always transmitted first. This is bit 0 of the value written to UART_THR.

Writes to the UART_THR register clear the THRE flag. Transfers of data from UART_THR to the transmit shift registers (TSR) set this status flag in UART_LSR again.

When enabled by the ETBEI bit in the UART_IER register, a 0 to 1 transition of the THRE flag requests an interrupt on the dedicated TXREQ output. This signal is routed through the DMA controller. If the associated DMA channel is enabled, the TXREQ signal functions as a DMA request, otherwise the

DMA controller simply forwards it to the system interrupt controller (SIC).

The UART_THR register and the internal TSR register can be seen as a two-stage transmit buffer. When data is pending in either one of these registers, the TEMT flag is low. As soon as the data has left the TSR register, the TEMT bit goes high again and indicates that all pending transmit operation has finished. At that time it is safe to disable the UCEN bit or to three-state off-chip line drivers.

UART Receive Operation

The receive operation uses the same data format as the transmit configuration, except that one valid stop bit is always sufficient. That is, the STB bit has no impact to the receiver.

After detection of the start bit, the received word is shifted into the internal shift register (RSR) at a bit rate of SCLK/(16 × Divisor). Once the appropriate number of bits (including one stop bit) is received, the content of the RSR register is transferred to the UART_RBR registers, shown in Figure 25-11 on page 25-27. Finally, the data ready (DR) bit and the status flags are updated in the UART_LSR register, to signal data reception, parity, and also error conditions, if required.

The RSR and the UART_RBR registers can be seen as almost a two-stage receive buffer. If the stop bit of a second byte is received before software reads the first byte from the UART_RBR register, an overrun error is reported and the first byte is overwritten.

If enabled by the ERBFI bit in the UART_IER register, a 0 to 1 transition of the DR flag requests an interrupt on the dedicated RXREQ output. This signal is routed through the DMA controller. If the associated DMA channel is enabled, the RXREQ signal functions as a DMA request, otherwise the DMA controller simply forwards it to the system interrupt controller.

If errors are detected during reception, an interrupt can be requested to a separate error interrupt output. This error request goes directly to the system interrupt controller. However, it is hard-wired with the error requests of other modules. The error handler routine may need to interrogate multiple modules as to whether they requested the event. Error requests must be enabled by the ELSI bit in the UART_IER register. The following error situations are detected. Every error has an indicating bit in the UART_LSR register.

- Overrun error (OE bit)
- Parity error (PE bit)
- Framing error/Invalid stop bit (FE bit)
- Break indicator (BI bit)

Reception is started when a falling edge is detected on the RX input pin. The receiver attempts to see a start bit. For better immunity against noise and hazards on the line, the receiver oversamples every bit 16 times and does a majority decision based on the middle three samples. The data is shifted immediately into the internal RSR register. After the 9th sample of the first stop bit is processed, the received data is copied to the UART_RBR register and the receiver recovers itself for further data.

The sampling clock, equal to 16 times the bit rate, samples the data bits close to their midpoint. Because the receiver clock is usually asynchronous to the transmitter's data rate, the sampling point may drift relative to the center of the data bits. The sampling point is synchronized again with each start bit, so the error accumulates only over the length of a single word. A receive filter removes spurious pulses of less than two times the sampling clock period.

IrDA Transmit Operation

To generate the IrDA pulse transmitted by the UART, the normal NRZ output of the transmitter is first inverted if the TPOLC bit is cleared, so a 0 is transmitted as a high pulse of 16 UART clock periods and a 1 is transmitted as a low pulse for 16 UART clock periods. The leading edge of the pulse is then delayed by six UART clock periods. Similarly, the trailing edge of the pulse is truncated by eight UART clock periods. This results in the final representation of the original 0 as a high pulse of only 3 clock periods out of 16 clock periods in the cycle. The pulse is centered around the middle of the bit time, as shown in Figure 25-3. The final IrDA pulse is fed to the off-chip infrared driver.

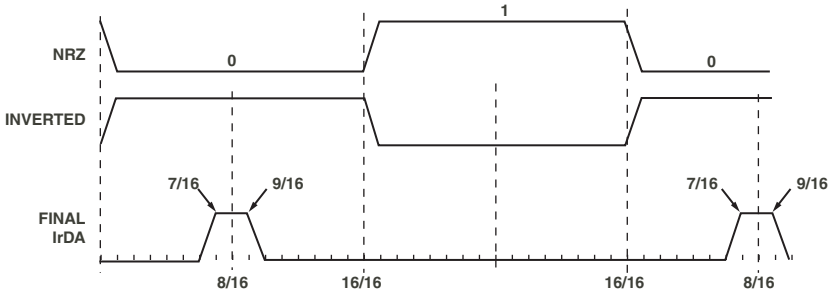


Figure 25-3. IrDA Transmit Pulse

This modulation approach ensures a pulse width output from the UART of three cycles high out of every 16 UART clock cycles. As shown in Table 25-1 on page 25-14, the error terms associated with the bit rate generator are very small and well within the tolerance of most infrared transceiver specifications.

IrDA Receive Operation

The IrDA receiver function is more complex than the transmit function. The receiver must discriminate the IrDA pulse and reject noise. To do

this, the receiver looks for the IrDA pulse in a narrow window centered around the middle of the expected pulse.

Glitch filtering is accomplished by counting 16 system clocks from the time an initial pulse is seen. If the pulse is absent when the counter expires, it is considered a glitch. Otherwise, it is interpreted as a 0. This is acceptable because glitches originating from on-chip capacitive cross-coupling typically do not last for more than a fraction of the system clock period. Sources outside of the chip and not part of the transmitter can be avoided by appropriate shielding. The only other source of a glitch is the transmitter itself. The processor relies on the transmitter to perform within specification. If the transmitter violates the specification, unpredictable results may occur. The 4-bit counter adds an extra level of protection at a minimal cost. Note that because the system clock can change across systems, the longest glitch tolerated is inversely proportional to the system clock frequency.

The receive sampling window is determined by a counter that is clocked at the 16× bit-time sample clock. The sampling window is re-synchronized with each start bit by centering the sampling window around the start bit.

The polarity of receive data is selectable, using the IRPOL bit. Figure 25-4 on page 25-11 gives examples of each polarity type.

- IRPOL = 0 assumes that the receive data input idles 0 and each active 1 transition corresponds to a UART NRZ value of 0.
- IRPOL = 1 assumes that the receive data input idles 1 and each active 0 transition corresponds to a UART NRZ value of 0.

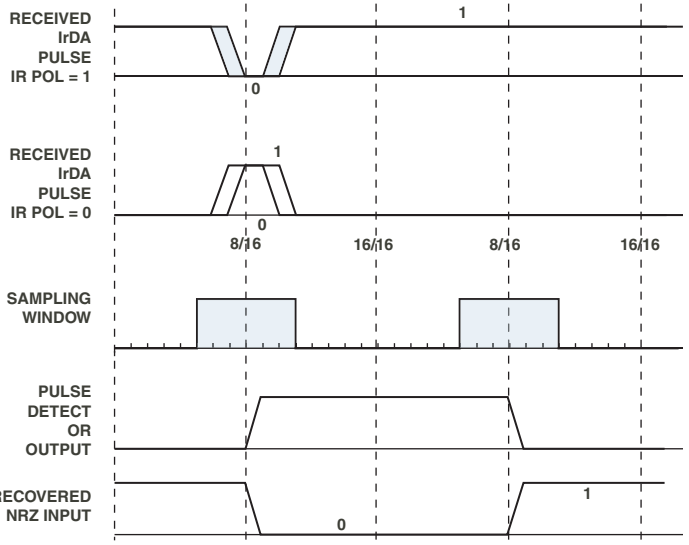


Figure 25-4. IrDA Receiver Pulse Detection

Interrupt Processing

Each UART module has three interrupt outputs. One is dedicated for transmission, one for reception, and the third is used to report line status. As shown in Figure 25-1 on page 25-3, the transmit and receive requests are routed through the DMA controller. The status request goes directly to the system interrupt controller after being ORed with interrupt signals from other modules.

If the associated DMA channel is enabled, the request functions as a DMA request. If the DMA channel is disabled, it simply forwards the request to the system interrupt controller. Note that a DMA channel must be associated with the UART module to enable TX and RX interrupts. Otherwise, the transmit and receive requests cannot be forwarded. Refer to the description of the peripheral map registers in the *Direct Memory Access* chapter.

Transmit interrupts are enabled by the ETBEI bit in the UART_IER register. If set, the transmit request is asserted when the THRE bit in the UART_LSR register transitions from 0 to 1, indicating that the TX buffer is ready for new data.

Note that the THRE bit resets to 1. When the ETBEI bit is set in the UART_IER register, the UART module immediately issues an interrupt or DMA request. In this way, no special handling of the first character is required when transmission of a string is initiated. Simply set the ETBEI bit and let the interrupt service routine load the first character from memory and write it to the UART_THR register in the normal manner. Accordingly, the ETBEI bit can be cleared if the string transmission has completed. For more information, see "DMA Mode" on page 25-18.

The THRE bit is cleared by hardware when new data is written to the UART_THR register. These writes also clear the TX interrupt request. However, they also initiate further transmission. If software doesn't want to continue transmission, the TX request can alternatively be cleared by either clearing the ETBEI bit or by reading the UART_IIR register.

Receive interrupts are enabled by the ERBFI bit in the UART_IER register. If set, the receive request is asserted when the DR bit in the UART_LSR register transitions from 0 to 1, indicating that new data is available in the UART_RBR register. When software reads the UART_RBR, hardware clears the DR bit again. Reading UART_RBR also clears the RX interrupt request.

Status interrupts are enabled by the ELSI bit in the UART_IER register. If set, the status interrupt request is asserted when any error bit in the

UART_LSR register transitions from 0 to 1. Refer to "UART Line Status (UART_LSR) Register" on page 25-25 for details. Reading the UART_LSR register clears the error bits destructively. These reads also clear the status interrupt request.

For legacy reasons, the UART_IIR registers still reflect the UART interrupt status. Legacy operation may require bundling all UART interrupt sources to a single interrupt channel and servicing them all by the same software routine. This can be established by globally assigning all UART interrupts to the same interrupt priority, by using the system interrupt controller.

If either the line status interrupt or the receive data interrupt has been assigned a lower interrupt priority by the system interrupt controller, a deadlock condition can occur. To avoid this, always assign the lowest priority of the enabled UART interrupts to the UART_THR empty event.

Bit Rate Generation

The UART clock is enabled by the UCEN bit in the UART_GCTL register.

The bit rate is characterized by the system clock (SCLK) and the 16-bit divisor. The divisor is split into the UART_DLL and the UART_DLH registers. These registers form a 16-bit divisor. The bit clock is divided by 16 so that:

bit rate = SCLK/(16 × divisor)
divisor = 65536 when UART_DLL = UART_DLH = 0

Table 25-1 provides example divide factors required to support most standard baud rates.

Table 25-1. UART Bit Rate Examples With 100 MHz SCLK

Bit Rate	DL	Actual	% Error
2400	2604	2400.15	0.006
4800	1302	4800.31	0.007
9600	651	9600.61	0.006
19200	326	19171.78	0.147
38400	163	38343.56	0.147
57600	109	57339.45	0.452
115200	54	115740.74	0.469
921600	7	892857.14	3.119
6250000	1	6250000	–

Careful selection of SCLK frequencies, that is, even multiples of desired bit rates, can result in lower error percentages.

Note that the UART module is clocked 16 times faster than the bit clock. This is required to oversample bits on reception and to generate RZI code in IrDA mode.

Autobaud Detection

At the chip level, the UART RX pin is routed to the alternate capture input (TACIx) of a general purpose timer. When working in WDTH_CAP mode this timer can be used to automatically detect the bit rate applied to the RX pin by an external device. For more information, see Chapter 9, “General-Purpose Ports”.

The capture capabilities of the timers are often used to supervise the bit rate at runtime. If the Blackfin UART talks to a device supplied by a weak

clock oscillator that drifts over time, the Blackfin can re-adjust its UART bit rate dynamically.

Often, autobaud detection is used for initial bit rate negotiations. In this case, the Blackfin processor is most likely a slave device waiting for the host to send a predefined autobaud character (see below). This is the scenario used for UART booting. In this scenario, the UART clock enable bit UCEN should not be enabled while autobaud detection is performed. This prevents the UART from starting reception with incorrect bit rate matching. Alternatively, the UART can be disconnected from the RX pin by setting the LOOP_ENA bit.

A software routine can detect the pulse widths of serial stream bit cells. Because the sample base of the timers is synchronous with the UART operation—all derived from SCLK—the pulse widths can be used to calculate the baud rate divider for the UART.

divisor = TIMER_WIDTH/(16 × number of captured UART bits)

In order to increase the number of timer counts and therefore the resolution of the captured signal, it is recommended not to measure just the pulse width of a single bit, but to enlarge the pulse of interest over more bits. Traditionally, a NULL character (ASCII 0x00) was used in autobaud detection, as shown in Figure 25-5.

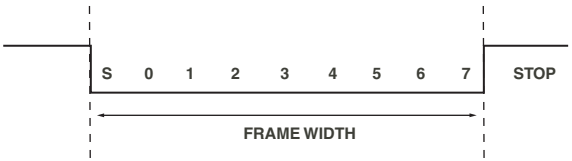


Figure 25-5. Autobaud Detection Character 0x00

Because the example frame in Figure 25-5 encloses 8 data bits and 1 start bit, apply the formula:

divisor = TIMER_WIDTH/(16 × 9)

Real UART RX signals often have asymmetrical falling and rising edges, and the sampling logic level is not exactly in the middle of the signal voltage range. At higher bit rates, such pulse width-based autobaud detection might not return adequate results without additional analog signal conditioning. Measuring signal periods works around this issue and is strongly recommended.

For example, predefine ASCII character "@" (0x40) as the autobaud detection character and measure the period between two subsequent falling edges. As shown in Figure 25-6, measure the period between the falling edge of the start bit and the falling edge after bit 6. Since this period encloses eight bits, apply the formula:

divisor = `TIMER_PERIOD`>>7

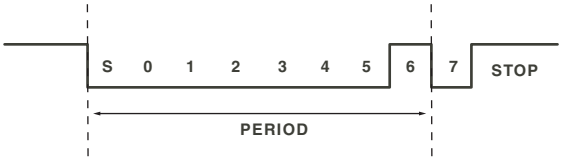


Figure 25-6. Autobaud Detection Character 0x40

An example is provided in Listing 25-2 on page 25-34.

Programming Model

The following sections describe a programming model for the UART.

Non-DMA Mode

In non-DMA mode, data is moved to and from the UART by the processor core. To transmit a character, load it into `UART_THR`. Received data can be read from `UART_RBR`. The processor must write and read one character at time.

To prevent any loss of data and misalignments of the serial datastream, the UART_LSR register provides two status flags for handshaking—THRE and DR.

The THRE flag is set when UART_THR is ready for new data and cleared when the processor loads new data into UART_THR. Writing UART_THR when it is not empty overwrites the register with the new value and the previous character is never transmitted.

The DR flag signals when new data is available in UART_RBR. This flag is cleared automatically when the processor reads from UART_RBR. Reading UART_RBR when it is not full returns the previously received value. When UART_RBR is not read in time, newly received data overwrites UART_RBR and the OE flag is set.

With interrupts disabled, these status flags can be polled to determine when data is ready to move. Note that because polling is processor intensive, it is not typically used in real-time signal processing environments. Be careful if transmit and receive are served by different software threads, because read operations on the UART_LSR and UART_IIR registers are destructive. Polling the SIC_ISR register without enabling the interrupts by SIC_MASK is an alternate method of operation to consider. Software can write up to two words into the UART_THR register before enabling the UART clock. As soon as the UCEN bit is set, those two words are sent.

Alternatively, UART writes and reads can be accomplished by interrupt service routines. Separate interrupt lines are provided for UART TX, UART RX, and UART error/status. The independent interrupts can be enabled individually by the UART_IER register.

The ISRs can evaluate the status bit field within the UART_IIR register to determine the signalling interrupt source. If more than one source is signalling, the status field displays the one with the highest priority. Interrupts also must be assigned and unmasked by the processor's interrupt controller. The ISRs must clear the interrupt latches explicitly. See Figure 25-13 on page 25-30.

DMA Mode

In this mode, separate receive (RX) and transmit (TX) DMA channels move data between the UART and memory. The software does not have to move data, it just has to set up the appropriate transfers either through the descriptor mechanism or through autobuffer mode.

DMA channels provide a 4-deep FIFO, resulting in total buffer capabilities of 6 words at both the transmit and receive sides. In DMA mode, the latency is determined by the bus activity and arbitration mechanism and not by the processor loading and interrupt priorities.

DMA interrupt routines must explicitly write "1" to the corresponding DMA_IRQ_STATUS registers to clear the latched request of the pending interrupt.

The UART's DMA is enabled by first setting up the system DMA control registers and then enabling the UART ERBFI and/or ETBEI interrupts in the UART_IER register. This is because the interrupt request lines double as DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. The UART's error interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.

For transmit DMA, it is recommended that the SYNC bit in the DMA_CONFIG register be set. With this bit set, the interrupt generation is delayed until the entire DMA FIFO has been drained to the UART module. The UART TX DMA interrupt service routine is allowed to start another DMA sequence or to clear the ETBEI control bit only when the SYNC bit is set.

If another DMA is started while data is still pending in the UART transmitter, there is no need to pulse the ETBEI bit to initiate the second DMA. If, however, the recent byte has already been loaded into the TSR registers

(that is, the THRE bit is set), then the ETBEI bit must be cleared and set again to let the second DMA start.

In DMA transmit mode, the ETBEI bit enables the peripheral request to the DMA FIFO. The strobe on the memory side is still enabled by the DMAEN bit. If the DMA count is less than the DMA FIFO depth, which is 4, then the DMA interrupt might be requested before the ETBEI bit is set. If this is not wanted, set the SYNC bit in the DMA_CONFIG register.

Regardless of the SYNC setting, the DMA stream has not left the UART transmitter completely at the time the interrupt is generated. If the UART clock was disabled without additional polling of the TEMT bit, transmission may abort in the middle of the stream—causing data loss.

The UART's DMA supports 8-bit and 16-bit operation, but not 32-bit operation. Sign extension is not supported.

Mixing Modes

Especially on the transmit side, switching from DMA mode to non-DMA operation on the fly requires some thought. By default, the interrupt timing of the DMA is synchronized with the memory side of the DMA FIFOs. The TX DMA completion interrupt is generated after the last byte has been copied from the memory into the DMA FIFO. The TX DMA interrupt service routine is not yet permitted to start other DMA sequences or to switch to non-DMA transmission. The interrupt is requested by the time the DMA_DONE bit is set. The DMA_RUN bit, however, remains set until the data has completely left the TX DMA FIFO.

Therefore, when planning to switch from DMA to non-DMA of operation, always set the SYNC bit in the DMA_CONFIG word of the last descriptor or work unit before handing over control to non-DMA mode. Then, after the interrupt occurs, software can write new data into the UART_THR register as soon as the THRE bit permits. If the SYNC bit cannot be set, software can poll the DMA_RUN bit instead.

When switching from non-DMA to DMA operation, take care that the very first DMA request is issued properly. If the DMA is enabled while the UART is still transmitting, no precaution is required. If, however, the DMA is enabled after the TEMT bit became high, the ETBEI bit should be pulsed to initiate DMA transmission.

UART Registers

The processor provides a set of PC-style industry-standard control and status registers for each UART. These memory-mapped registers (MMRs) are byte-wide registers that are mapped as half words with the most significant byte zero filled. Table 25-2 on page 25-20 provides an overview of the UART registers.

Consistent with industry-standard devices, multiple registers are mapped to the same address location. The UART_DLH and UART_DLL registers share their addresses with the UART_THR registers, the UART_RBR registers, and the UART_IER registers. The DLAB bit in the UART_LCR register controls which set of registers is accessible at a given time. Software must use 16-bit word load/store instructions to access these registers.

Transmit and receive channels are both buffered. The UART_THR registers buffer the transmit shift register (TSR) and the UART_RBR registers buffer the receive shift register (LSR). The shift registers are not directly accessible by software.

Table 25-2. UART Register Overview

Name	Address Offset	DLAB Bit Setting	Operation	Reset Value	Function
UART_RBR	0x0000	0	R	0x00	Receive buffer register
UART_THR	0x0000	0	W	0x00	Transmit holding register
UART_DLL	0x0000	1	R/W	0x01	Divisor latch low byte

Table 25-2. UART Register Overview

Name	Address Offset	DLAB Bit Setting	Operation	Reset Value	Function
UART_IER	0x0004	0	R/W	0x00	Interrupt enable register
UART_DLH	0x0004	1	R/W	0x00	Divisor latch high byte
UART_IIR	0x0008	X	R Read operations are destructive	0x01	Interrupt identification register
UART_LCR	0x000C	X	R/W	0x00	Line control register
UART_MCR	0x0010	X	R/W	0x00	Modem control register
UART_LSR	0x0014	X	R Read operations are destructive	0x60	Line status register
UART_SCR	0x001C	X	R/W	0x00	Scratch register
UART_GCTL	0x0024	X	R/W	0x00	Global control register

UART Line Control (UART_LCR) Register

The UART_LCR register, shown in Figure 25-7, controls the format of received and transmitted character frames.

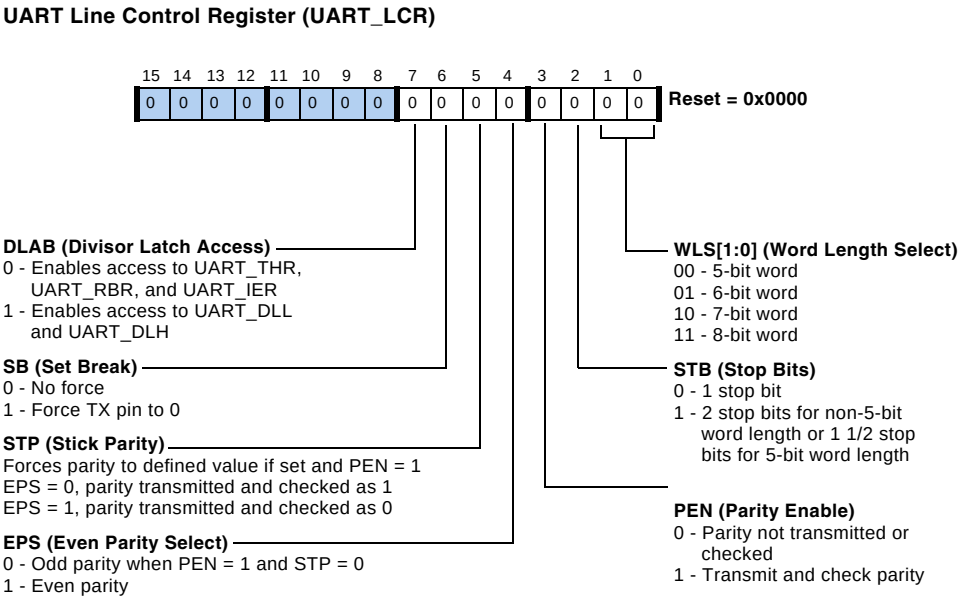

UART Line Control Register (UART_LCR)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DLAB (Divisor Latch Access)
0 - Enables access to UART_THR, UART_RBR, and UART_IER
1 - Enables access to UART_DLL and UART_DLH
SB (Set Break)
0 - No force
1 - Force TX pin to 0
STP (Stick Parity)
Forces parity to defined value if set and PEN = 1
EPS = 0, parity transmitted and checked as 1
EPS = 1, parity transmitted and checked as 0
EPS (Even Parity Select)
0 - Odd parity when PEN = 1 and STP = 0
1 - Even parity
WLS[1:0] (Word Length Select)
00 - 5-bit word
01 - 6-bit word
10 - 7-bit word
11 - 8-bit word
STB (Stop Bits)
0 - 1 stop bit
1 - 2 stop bits for non-5-bit word length or 1 1/2 stop bits for 5-bit word length
PEN (Parity Enable)
0 - Parity not transmitted or checked
1 - Transmit and check parity

Figure 25-7. UART Line Control Register

The 2-bit WLS field determines whether the transmitted and received UART word consists of 5, 6, 7 or 8 data bits.

The STB bit controls how many stop bits are appended to transmitted data. When STB = 0, one stop bit is transmitted. If WLS is non zero, STB = 1 instructs the transmitter to add one additional stop bit, two stop bits in total. If WLS = 0 and 5-bit operation is chosen, STB = 1 forces the transmitter to append one additional half bit, 1½ stop bits in total. Note that this bit does not impact data reception—the receiver is always satisfied with one stop bit.

The PEN bit inserts one additional bit between the most significant data bit and the first stop bit. The polarity of this so-called parity bit depends on data and the STP and EPS control bits. Both transmitter and receiver calculate the parity value. The receiver compares the received parity bit with the expected value and issues a parity error if they don't match. If PEN is cleared, the STP and the EPS bits are ignored.

The STP bit controls whether the parity is generated by hardware based on the data bits or whether it is set to a fixed value. If STP = 0 the hardware calculates the parity bit value based on the data bits. Then, the EPS bit determines whether odd or even parity mode is chosen. If EPS = 0, odd parity is used. That means that the total count of logical-1 data bits including the parity bit must be an odd value. Even parity is chosen by STP = 0 and EPS = 1. Then, the count of logical-1 bits must be a even value. If the STP bit is set, then hardware parity calculation is disabled. In this case, the sent and received parity equals the inverted EPS bit. The example in Table 25-3 summarizes polarity behavior assuming 8-bit data words (WLS = 3).

Table 25-3. UART Parity

PEN	STP	EPS	Data (hex)	Data (binary, LSB first)	Parity
0	x	x	x	x	None
1	0	0	0x60	0000 0110	1
1	0	0	0x57	1110 1010	0
1	0	1	0x60	0000 0110	0
1	0	1	0x57	1110 1010	1
1	1	0	x	x	1
1	1	1	x	x	0

If set, the SB bit forces the TX pin to low asynchronously, regardless of whether or not data is currently transmitted. It functions even when the

UART clock is disabled. Since the TX pin normally drives high, it can be used as a flag output pin, if the UART is not used.

The DLAB bit controls whether the UART_RBR, UART_THR and UART_IER registers are accessible by the peripheral bus (DLAB = 0) or the divisor latch registers UART_DLH and UART_DLL alternatively (DLAB = 1).

UART Modem Control (UART_MCR) Register

The UART_MCR register controls the UART port, as shown in Figure 25-8. Even if modem functionality is not supported, the UART_MCR register is available in order to support the loopback mode.

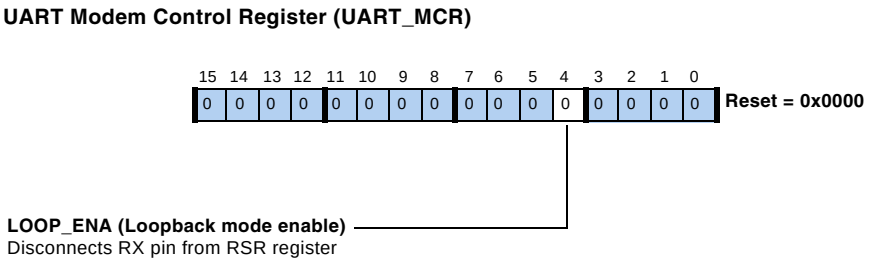

UART Modem Control Register (UART_MCR)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
LOOP_ENA (Loopback mode enable)
Disconnects RX pin from RSR register

Figure 25-8. UART Modem Control Registers

Loopback mode disconnects the receiver's input from the RX pin, but redirects it to the transmit output internally.

UART Line Status (UART_LSR) Register

The UART_LSR register contains UART status information as shown in Figure 25-9.

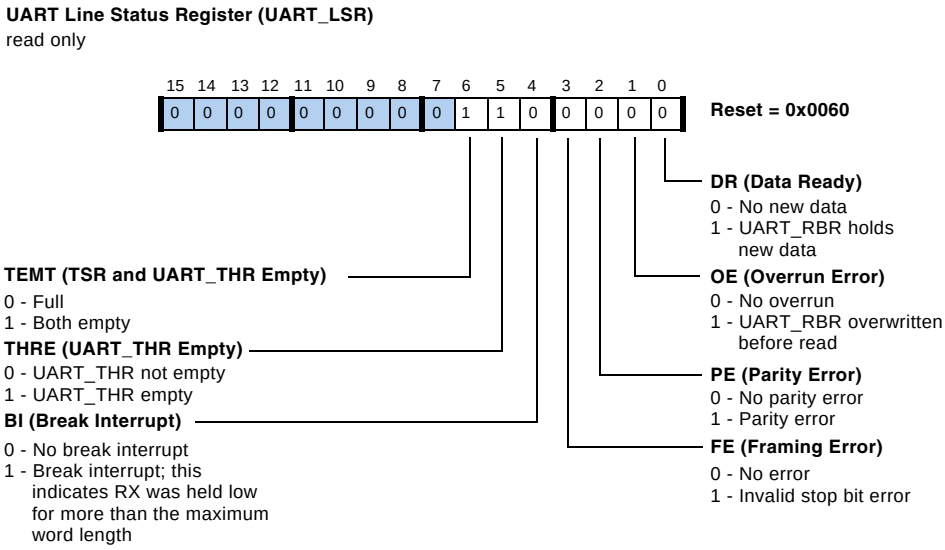

UART Line Status Register (UART_LSR)
read only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0
Reset = 0x0060
DR (Data Ready)
0 - No new data
1 - UART_RBR holds new data
OE (Overrun Error)
0 - No overrun
1 - UART_RBR overwritten before read
PE (Parity Error)
0 - No parity error
1 - Parity error
FE (Framing Error)
0 - No error
1 - Invalid stop bit error
TEMT (TSR and UART_THR Empty)
0 - Full
1 - Both empty
THRE (UART_THR Empty)
0 - UART_THR not empty
1 - UART_THR empty
BI (Break Interrupt)
0 - No break interrupt
1 - Break interrupt; this indicates RX was held low for more than the maximum word length

Figure 25-9. UART Line Status Register

The DR bit indicates that data is available in the receiver and can be read from the UART_RBR register. The bit is set by hardware when the receiver detects the first valid stop bit. It is cleared by hardware when the UART_RBR register is read.

The OE bit indicates that a start bit condition has been detected, but the internal receive shift register (RSR) and the receive buffer (UART_RBR) already contain data. New data overwrites the content of the buffers. To avoid overruns read the UART_RBR register in time. The OE bit cleared when the UART_LSR register is read.

The PE bit indicates that the received parity bit does not match the expected value. The PE bit is set simultaneously with the DR bit. The PE bit

cleared when the UART_LSR register is read. Invalid parity bits can be simulated by setting the FPE bit in the UART_GCTL register.

The FE bit indicates that the first stop bit has been sampled low. It is cleared by hardware when the UART_RBR register is read. Invalid stop bits can be simulated by setting the FFE bit in the UART_GCTL register.

The BI bit indicates that the first stop bit has been sampled low and the entire data word, including parity bit, consists of low bits only. It is cleared by hardware when the UART_RBR register is read.

Because of the destructive nature of these read operations, special care should be taken. For more information, see the *Memory* chapter of the *ADSP-BF52x Blackfin Processor Hardware Reference*.

The THRE bit indicates that the UART transmit channel is ready for new data and software can write to UART_THR. Writes to UART_THR clear the THRE bit. It is set again when data is passed from UART_THR to the internal TSR register.

The TEMT bit indicates that both the UART_THR register and the internal TSR register are empty. In this case the program is permitted to write to the UART_THR register twice without losing data. The TEMT bit can also be used as an indicator that pending UART transmission has been completed. At that time it is safe to disable the UCEN bit or to three-state the off-chip line driver.

UART Transmit Holding (UART_THR) Register

The write-only UART_THR register, shown in Figure 25-10, is mapped to the same address as the read-only UART_RBR and UART_DLL registers. To access UART_THR, the DLAB bit in UART_LCR must be cleared. When the DLAB

bit is cleared, writes to this address target the UART_THR register, and reads from this address return the UART_RBR register.

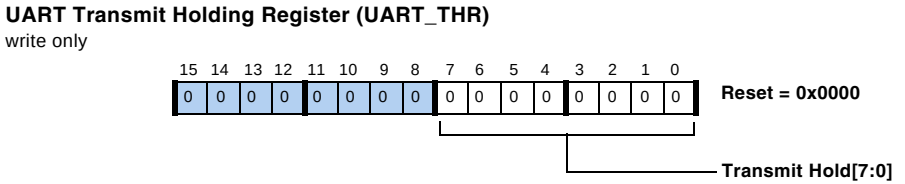

UART Transmit Holding Register (UART_THR)
write only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Transmit Hold[7:0]

Figure 25-10. UART Transmit Holding Register

UART Receive Buffer (UART_RBR) Register

The read-only UART_RBR register, shown in Figure 25-11, is mapped to the same address as the write-only UART_THR and UART_DLL registers. To access UART_RBR, the DLAB bit in UART_LCR must be cleared. When the DLAB bit is cleared, writes to this address target the UART_THR register, while reads from this address return the value in the UART_RBR register.

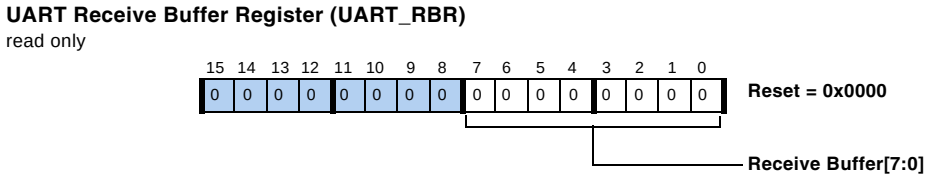

UART Receive Buffer Register (UART_RBR)
read only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Receive Buffer[7:0]

Figure 25-11. UART Receive Buffer Register

UART Interrupt Enable (UART_IER) Register

The UART_IER register, shown in Figure 25-12 on page 25-28, is used to enable requests for system handling of empty or full states of UART data registers. Unless polling is used as a means of action, the ERBFI and/or ETBEI bits in this register are normally set.

Setting this register without enabling system DMA causes the UART to notify the processor of data inventory state by means of interrupts. For proper operation in this mode, system interrupts must be enabled, and appropriate interrupt handling routines must be present. For backward compatibility, the UART_IIR still reflects the correct interrupt status.

Each UART features three separate interrupt channels to handle data transmit, data receive, and line status events independently, regardless of whether DMA is enabled or not. On some processors, the status interrupt channels from multiple UARTs may be ORed prior to being connected to the system interrupt controller. See Chapter 5, "System Interrupts" for more information.

With system DMA enabled, the UART uses DMA to transfer data to or from the processor. Dedicated DMA channels are available to receive and transmit operation. Line error handling can be configured completely independently from the receive/transmit setup.

The UART_IER registers are mapped to the same address as the UART_DLH registers. To access UART_IER, the DLAB bit in UART_LCR must be cleared.

UART Interrupt Enable Register (UART_IER)

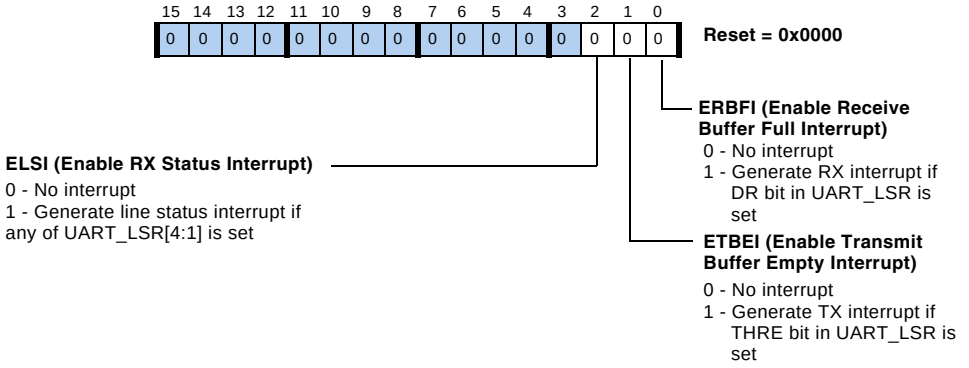

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
ERBFI (Enable Receive Buffer Full Interrupt)
0 - No interrupt
1 - Generate RX interrupt if DR bit in UART_LSR is set
ELSI (Enable RX Status Interrupt)
0 - No interrupt
1 - Generate line status interrupt if any of UART_LSR[4:1] is set
ETBEI (Enable Transmit Buffer Empty Interrupt)
0 - No interrupt
1 - Generate TX interrupt if THRE bit in UART_LSR is set

Figure 25-12. UART Interrupt Enable Register

The UART's DMA is enabled by first setting up the system DMA control registers and then enabling the UART ERBFI and/or ETBEI interrupts in the UART_IER register. This is because the interrupt request lines double as DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. However, UART's error interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.

The ELSI bit enables interrupt generation on an independent interrupt channel when any of the following conditions are raised by the respective bit in the UART_LSR register:

- Receive overrun error (OE)
- Receive parity error (PE)
- Receive framing error (FE)
- Break interrupt (BI)

UART Interrupt Identification (UART_IIR) Register

The UART_IIR register conveys interrupt status within the UART. When cleared, the NINT bit signals that an interrupt is pending. The STATUS field indicates the highest priority pending interrupt. The receive line status has the highest priority; the UART_THR empty interrupt has the lowest priority. In the case where both interrupts are signaling, the UART_IIR reads 0x06.

When a UART interrupt is pending, the interrupt service routine needs to clear the interrupt latch explicitly. Figure 25-13 shows how to clear any of the three latches.

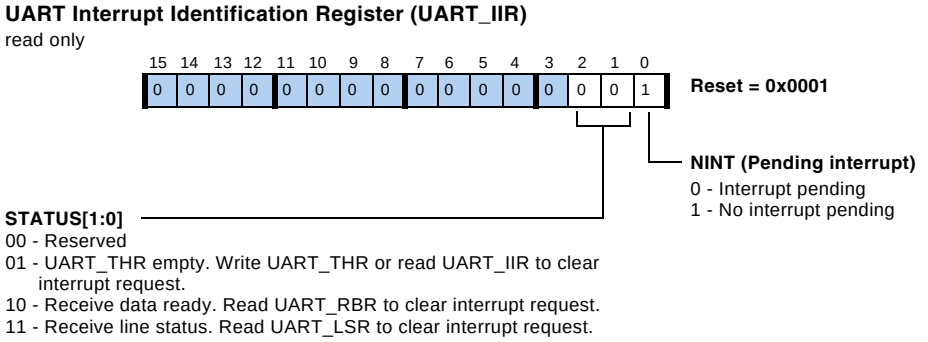

UART Interrupt Identification Register (UART_IIR)
read only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
Reset = 0x0001
NINT (Pending interrupt)
0 - Interrupt pending
1 - No interrupt pending
STATUS[1:0]
00 - Reserved
01 - UART_THR empty. Write UART_THR or read UART_IIR to clear interrupt request.
10 - Receive data ready. Read UART_RBR to clear interrupt request.
11 - Receive line status. Read UART_LSR to clear interrupt request.

Figure 25-13. UART Interrupt Identification Register

The TX interrupt request is cleared by writing new data to the UART_THR register or by reading the UART_IIR register. Please note the special role of the UART_IIR register read in the case where the service routine does not want to transmit further data.

If software stops transmission, it must read the UART_IIR register to reset the interrupt request. As long as the UART_IIR register reads 0x04 or 0x06 (indicating that another interrupt of higher priority is pending), the UART_THR empty latch cannot be cleared by reading UART_IIR.

Because of the destructive nature of these read operations, special care should be taken. For more information, see the *Memory* chapter of the *ADSP-BF52x Blackfin Processor Hardware Reference*.

UART Divisor Latch (UART_DLL and UART_DLH) Registers

The UART_DLL register is mapped to the same address as the UART_THR and UART_RBR registers. The UART_DLH register is mapped to the same address as

the UART_IER register. The DLAB bit in UART_LCR must be set before the UART_DLL and UART_DLH registers, shown in Figure 25-14, can be accessed.

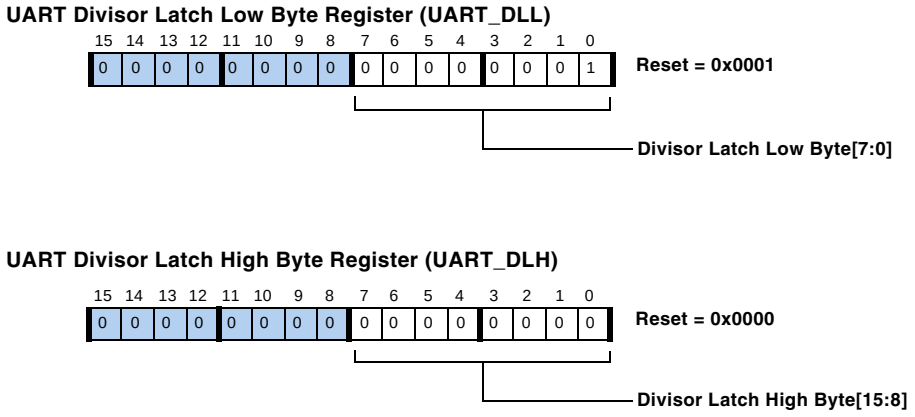

UART Divisor Latch Low Byte Register (UART_DLL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
Reset = 0x0001
Divisor Latch Low Byte[7:0]
UART Divisor Latch High Byte Register (UART_DLH)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Divisor Latch High Byte[15:8]

Figure 25-14. UART Divisor Latch Registers

Note the 16-bit divisor formed by UART_DLH and UART_DLL resets to 0x0001, resulting in the highest possible clock frequency by default. If the UART is not used, disabling the UART clock saves power. The UART_DLH and UART_DLL registers can be programmed by software before or after setting the UCEN bit.

UART Scratch (UART_SCR) Register

The 8-bit UART_SCR register, shown in Figure 25-15, is used for general-purpose data storage and does not control the UART hardware in any way. The contents are reset to 0x00.

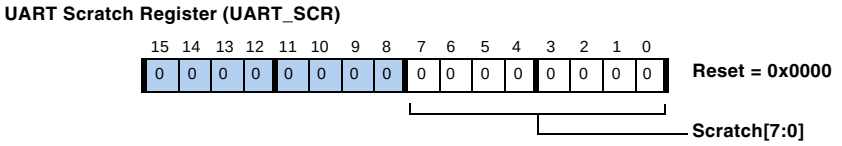


Figure 25-15. UART Scratch Register

UART Global Control (UART_GCTL) Register

The UART_GCTL register, shown in Figure 25-16, contains the enable bit for internal UART clocks and for the IrDA mode of operation of the UART.

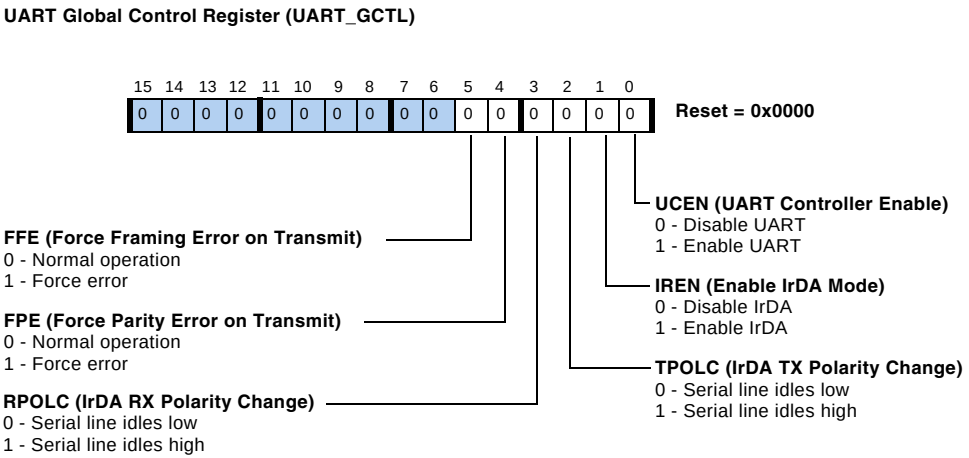

UART Global Control Register (UART_GCTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
FFE (Force Framing Error on Transmit)
0 - Normal operation
1 - Force error
FPE (Force Parity Error on Transmit)
0 - Normal operation
1 - Force error
RPOLC (IrDA RX Polarity Change)
0 - Serial line idles low
1 - Serial line idles high
UCEN (UART Controller Enable)
0 - Disable UART
1 - Enable UART
IREN (Enable IrDA Mode)
0 - Disable IrDA
1 - Enable IrDA
TPOLC (IrDA TX Polarity Change)
0 - Serial line idles low
1 - Serial line idles high

Figure 25-16. UART Global Control Register

The UCEN bit enables the UART clocks. It also resets the state machine and control registers when cleared.

This bit has been introduced to save power if the UART is not used. When porting code, be sure to enable this bit.

The IrDA TX polarity change bit and the IrDA RX polarity change bit are effective only in IrDA mode. The two force error bits, FPE and FFE, are intended for test purposes. They are useful for debugging software, especially in loopback mode.

Programming Examples

The subroutine in Listing 25-1 shows a typical UART initialization sequence.

Listing 25-1. UART Initialization

```
/*************************************************************
 * Configures UART in 8 data bits, no parity, 1 stop bit mode.
 * Input parameters: r0 holds divisor latch value to be
 *                   written into
 *                   DLH:DLL registers.
 *                   p0 contains the UART_GCTL register address
 *  Return values:    none
 *************************************************************/
uart_init:
   [--sp] = r7;
   r7 = UCEN (z);  /* First of all, enable UART clock */
   w[p0+UART0_GCTL-UART0_GCTL] = r7;

   r7 = DLAB (z);                    /* to set bit rate    */
   w[p0+UART0_LCR-UART0_GCTL] = r7;   /* set DLAB bit first */
   w[p0+UART0_DLL-UART0_GCTL] = r0; /* write lower byte to DLL */
```

```
   r7 = r0 >> 8;
   w[p0+UART0_DLH-UART0_GCTL] = r7; /* write upper byte to DLH */

   r7 = STB | WLS(8) (z);    /* clear DLAB again and config to */
   w[p0+UART0_LCR-UART0_GCTL] = r7;
                            /* 8 bits, no parity, 2 stop bits */

   r7 = [sp++];
   rts;
uart_init.end:
```

The subroutine in Listing 25-2 performs autobaud detection similarly to UART boot.

Listing 25-2. UART Autobaud Detection Subroutine

```
/*************************************************************
 *  Assuming 8 data bits, this functions expects a '@'
 *  (ASCII 0x40) character
 *  on the UART RX pin. A Timer performs the autobaud detection.
 *  Input parameters: p0 contains the UART_GCTL register address
 *                    p1 contains the TIMER_CONFIG register
 *  address
 *  Return values:    r0 holds timer period value (equals 8 bits)
*************************************************************/
uart_autobaud:
   [--sp] = (r7:5,p5:5);
   r5.h = hi(TIMER0_CONFIG); /* for generic timer use calculate
*/
   r5.l = lo(TIMER0_CONFIG);   /* specific bits first */
   r7 = p1;
   r7 = r7 - r5;
   r7 >>= 4;         /* r7 holds the 'x' of TIMERx_CONFIG now */
   r5 = TIMEN0 (z);
   r5 <<= r7;        /* r5 holds TIMENx/TIMDISx now */
```

```
   r6 = TRUN0 | TOVL_ERR0 | TIMIL0 (z);
   r6 <<= r7;
   CC = r7 <= 3;
   r7 = r6 << 12;
   if !CC r6 = r7;   /* r6 holds TRUNx | TOVL_ERRx | TIMILx */

   p5.h = hi(TIMER_STATUS);
   p5.l = lo(TIMER_STATUS);
   w[p5 + TIMER_DISABLE - TIMER_STATUS] = r5; /* disable Timer x
*/
   [p5 + TIMER_STATUS - TIMER_STATUS] = r6;
               /* clear pending latches */
               /* period capture, falling edge to falling edge */
   r7 = TIN_SEL | IRQ_ENA | PERIOD_CNT | WDTH_CAP (z);
   w[p1 + TIMER0_CONFIG - TIMER0_CONFIG] = r7;
   w[p5+TIMER_ENABLE-TIMER_STATUS] = r5;

uart_autobaud.wait:   /* wait for timer event */
         r7 = w[p5 + TIMER_STATUS - TIMER_STATUS] (z);
         r7 = r7 & r5;
         CC = r7 == 0;
         if CC jump uart_autobaud.wait;
   w[p5 + TIMER_DISABLE - TIMER_STATUS] = r5; /* disable Timer x
*/
   [p5 + TIMER_STATUS - TIMER_STATUS] = r6;
                              /* clear pending latches */
   /* Save period value to R0 */
   r0 = [p1 + TIMER0_PERIOD - TIMER0_CONFIG];

   /* delay processing as autobaud character is still ongoing */
   r7 = OUT_DIS | IRQ_ENA | PERIOD_CNT | PWM_OUT (z);
   w[p1 + TIMER0_CONFIG - TIMER0_CONFIG] = r7;
   w[p5 + TIMER_ENABLE - TIMER_STATUS] = r5;
```

```
uart_autobaud.delay:
         r7 = w[p5 + TIMER_STATUS - TIMER_STATUS] (z);
         r7 = r7 & r5;
         CC = r7 == 0;
         if CC jump uart_autobaud.delay;
   w[p5 + TIMER_DISABLE - TIMER_STATUS] = r5;
   [p5 + TIMER_STATUS - TIMER_STATUS] = r6;
   (r7:5,p5:5) = [sp++];
   rts;
uart_autobaud.end:
```

The parent routine in Listing 25-3 performs autobaud detection, using as an example a processor whose TIMER4 is mapped to UART0 for this purpose. Note also that this example assumes the processor's UART0 pins are mapped to PORT G (PG7 and PG8).

Listing 25-3. UART Autobaud Detection Parent Routine

```
   p0.l = lo(PORTG_FER);
            /* function enable on UART0 pins PG7 and PG8 */
   p0.h = hi(PORTG_FER);
   r0 = PG7 | PG8 (z)
   w[p0] = r0;
   p0.l = lo(PORTG_MUX);
   p0.h = hi(PORTG_MUX);
   r0.l = 0x0020;
   r0.h = 0x0000;
   w[p0] = r0;
   p0.l = lo(UART0_GCTL);       /* select UART 0 */
   p0.h = hi(UART0_GCTL);
   p1.l = lo(TIMER4_CONFIG);    /* select TIMER 4 */
   p1.h = hi(TIMER4_CONFIG);
   call uart_autobaud;
   r0 >>= 7;       /* divide PERIOD value by (16 x 8) */
```

```
   call uart_init;
   ...
```

The subroutine in Listing 25-4 on page 25-37 transmits a character by polling operation.

Listing 25-4. UART Character Transmission

```
/*******************************************************
 *  Transmit a single byte by polling the THRE bit.
 *  Input parameters: r0 holds the character to be transmitted
 *                    p0 contains UART_GCTL register address
 *  Return values: none
*******************************************************/
uart_putc:
   [--sp] = r7;
uart_putc.wait:
         r7 = w[p0+UART0_LSR-UART0_GCTL] (z);
         CC = bittst(r7, bitpos(THRE));
         if !CC jump uart_putc.wait;
   w[p0+UART0_THR-UART0_GCTL] = r0; /* write initiates transfer
*/
   r7 = [sp++];
   rts;
uart_putc.end:
```

Use the routine shown in Listing 25-5 to transmit a C-style string that is terminated by a null character.

Listing 25-5. UART String Transmission

```
/*************************************************************
 *  Transmit a null-terminated string.
 *  Input parameters: p1 points to the string
 *                    p0 contains UART_GCTL register address
```

```
 *  Return values: none
*************************************************************/
uart_puts:
   [--sp] = rets;
   [--sp] = r0;
uart_puts.loop:
         r0 = b[p1++] (z);
         CC = r0 == 0;
         if CC jump uart_puts.exit;
         call uart_putc;
         jump uart_puts.loop;
uart_puts.exit:
   r0 = [sp++];
   rets = [sp++];
   rts;
uart_puts.end:
```

Note that polling the UART0_LSR register for transmit purposes may clear the receive error latch bits. It is, therefore, not recommended to poll UART0_LSR for transmission this way while data is being received. In that case, write a polling loop that reads UART_LSR once and then evaluates *all* status bits of interest, as shown in Listing 25-6.

Listing 25-6. UART Polling Loop

```
uart_loop:
   r7 = w[p0+UART0_LSR-UART0_GCTL] (z);
   CC = bittst(r7, bitpos(DR));
   if !CC jump uart_loop.transmit;
   r6 = w[p0+UART0_RBR-UART0_GCTL] (z);
   r5 = BI | OE | FE | PE (z);
   r5 = r5 & r7;
   CC = r5 == 0;
   if !CC jump uart_loop.error;
   b[p1++] = r6;         /* store byte */
```

```
uart_loop.transmit:
   CC = bittst(r7, bitpos(THRE));
   if !CC jump uart_loop;
   r5 = b[p2++] (z);     /* load next byte */
   w[p0+UART0_THR-UART0_GCTL] = r5;
   jump uart_loop;
uart_loop.error:
   ...
   jump uart_loop;
```

In non-DMA interrupt operation, the three UART interrupt request lines may or may not be ORed together in the system interrupt controller. If they had three different service routines, they may look as shown in Listing 25-7.

Listing 25-7. UART Non-DMA Interrupt Operation

```
isr_uart_rx:
   [--sp] = astat;
   [--sp] = r7;
   r7 = w[p0+UART0_RBR-UART0_GCTL] (z);
   b[p4++] = r7;
   ssync;
   r7 = [sp++];
   astat = [sp++];
   rti;
isr_uart_rx.end:

isr_uart_tx:
   [--sp] = astat;
   [--sp] = r7;
   r7 = b[p3++] (z);
   CC = r7 == 0;
   if CC jump isr_uart_tx.final;
   w[p0+UART0_THR-UART0_GCTL] = r7;
```

```
    r7 = [sp++];
    astat = [sp++];
    ssync;
    rti;
isr_uart_tx.final:
    r7 = w[p0+UART0_IER-UART0_GCTL] (z);
                /* clear TX interrupt enable */
    bitclr(r7, bitpos(ETBEI));   /* ensure this sequence is not */
    w[p0+UART0_IER-UART0_GCTL] = r7;
                /* interrupted by other IER accesses */
    ssync;
    r7 = [sp++];
    astat = [sp++];
    rti;
isr_uart_tx.end:

isr_uart_error:
    [--sp] = astat;
    [--sp] = r7;
    r7 = w[p0+UART0_LSR-UART0_GCTL] (z);
              /* read clears interrupt request */
              /* do something with the error */
    r7 = [sp++];
    astat = [sp++];
    ssync;
    rti;
isr_uart_error.end:
```

Listing 25-8 transmits a string by DMA operation, waits until DMA completes and sends an additional string by polling. Note the importance of the SYNC bit.

Listing 25-8. UART Transmission SYNC Bit Use

```
.section data;
```

```
.byte sHello[] = 'Hello Blackfin User',13,10,0;
.byte sWorld[] = 'How is life?',13,10,0;

.section program;
   ...
   p1.l = lo(IMASK);
   p1.h = hi(IMASK);
   r0.l = lo(isr_uart_tx);      /* register service routine */
   r0.h = hi(isr_uart_tx); /*Assume UART0 TX defaults to IVG10*/
   r0 = [p1 + IMASK - IMASK];   /* unmask interrupt in CEC */
   bitset(r0, bitpos(EVT_IVG10));
   [p1] = r0;
   p1.l = lo(SIC_IMASK0);
   p1.h = hi(SIC_IMASK0);
/* unmask interrupt in SIC */
/* (assume SIC_IMASK0 for this example)*/
   r0.l = 0x0080;
   r0.h = 0x0000;
   [p1] = r0;
   [--sp] = reti;    /* enable nesting of interrupts */

   p5.l = lo(DMA9_CONFIG);
/* setup DMA in STOP mode */
/* (assume DMA channel 9 for this example)*/
   p5.h = hi(DMA9_CONFIG);
   r7.l = lo(sHello);
   r7.h = hi(sHello);
   [p5+DMA9_START_ADDR-DMA9_CONFIG] = r7;
   r7 = length(sHello) (z);
   r7+= -1;           /* do not send trailing null character */
   w[p5+DMA9_X_COUNT-DMA9_CONFIG] = r7;
   r7 = 1;
   w[p5+DMA9_X_MODIFY-DMA9_CONFIG] = r7;
   r7 = FLOW_STOP | WDSIZE_8 | DI_EN | SYNC | DMAEN (z);
```

```
   w[p5] = r7;

   p0.l = lo(UART0_GCTL);   /* select UART 0 */
   p0.h = hi(UART0_GCTL);
   r0 = ETBEI (z);          /* enable and issue first request */
   w[p0+UART0_IER-UART0_GCTL] = r0;

wait4dma:  /* just one way to synchronize with the service rou-
tine */
   r0 = w[p5+DMA9_IRQ_STATUS-DMA9_CONFIG] (z);
   CC = bittst(r0,bitpos(DMA_RUN));
   if CC jump wait4dma;
   p1.l=lo(sWorld);
   p1.h=hi(sWorld);
   call uart_puts;

forever: jump forever;

isr_uart_tx:
   [--sp] = astat;
   [--sp] = r7;
   r7 = DMA_DONE (z);   /* W1C interrupt request */
   w[p5+DMA9_IRQ_STATUS-DMA9_CONFIG] = r7;
   r7 = 0;              /* pulse ETBEI for general case */
   w[p0+UART0_IER-UART0_GCTL] = r7;
   ssync;
   r7 = [sp++];
   astat = [sp++];
   rti;
isr_uart_tx.end:
```

Unique Information for the ADSP-BF52x Processor

None.

26 USB OTG CONTROLLER

This chapter describes the 6-pin USB OTG interface for the USB OTG controller.

This chapter includes the following sections:

- "Overview" on page 26-1
- "Interface Overview" on page 26-3
- "Description of Operation" on page 26-12
- "Functional Description" on page 26-51
- "Programming Model" on page 26-54
- "USB OTG Registers" on page 26-95
- "References" on page 26-148
- "Glossary of USB Terms " on page 26-148

Overview

The USB OTG controller provides a low-cost connectivity solution for consumer mobile devices such as cell phones, digital still cameras and MP3 players, allowing these devices to transfer data using a point-to-point USB connection without the need for a personal computer host.

The USB controller can operate in a traditional USB peripheral-only mode as well as the host mode presented in the On-The-Go (OTG) sup-

plement[1] to the USB 2.0 Specification[2]. In host mode, the USB module supports transfers at high-speed (480Mbps), full-speed (12Mbps), and low-speed (1.5Mbps) rates. Peripheral mode supports the high- and full-speed transfer rates.

The USB controller uses a peripheral bus slave interface to access its control and status registers as well as read and write to the endpoint packet buffers. Data is transferred to and from the USB controller through any of the seven transmit and seven receive endpoint FIFOs, EP1 – EP7, providing a total of 14 data endpoints. A DCB/DEB bus master interface provides eight DMA channels to provide a more efficient means of transferring large amounts of data between the controller and the Blackfin processor's memory map.

Features

The USB controller provides the following features:

- low speed, full speed, high speed rates supported
- one bidirectional control endpoint
- seven transmit and seven receive unidirectional endpoints
- 7.232K Bytes of FIFOs for packet buffering
- eight DMA master channels
- three top-level maskable general purpose interrupts
- one asynchronous wakeup interrupt
- VBUS control interrupts for external analog VBUS control

---

[1] On-The-Go Supplement to the USB 2.0 Specification, Rev 1.0a; June 24, 2003; USB-IF

[2] Universal Serial Bus Specification 2.0

- software-controlled clock control on each endpoint for power reduction
- session request protocol (SRP) and host negotiation protocol (HNP) capability
- host transaction scheduling in hardware
- soft connect/disconnect feature
- full- and high-speed physical layer UTMI+ level 2 interface for on-chip PHY
- backwards compatible with existing USB 1.1 hosts

The number of active endpoints at one time is only limited by device requirements or system bandwidth, because each endpoint operates independently from the next. The maximum buffer size per endpoint is 1024 bytes. Software determines the type of transfer for each endpoint individually and also the manner in which it is transferred between the USB controller and memory (DMA or interrupt-based). Endpoint zero is used solely for receive and transmit control transfers, which are used for device configuration and information gathering.

Interface Overview

The USB controller operates in either of two USB operation modes (peripheral or host mode) at a given time.

In peripheral mode, the USB controller encodes, decodes, checks, and directs all USB packets sent and received, responding appropriately to host requests. Data is transferred from the processor core memory into the device's TX FIFOs to be transmitted onto USB as IN packets. In the other direction USB OUT packets are received into the RX FIFOs (having been sent from the host) and transferred to system memory for processing or storage. In peripheral mode, the USB controller acts as a slave device to

another USB host; either a personal computer or another OTG host controller.

When operating in host mode, the USB controller uses simple hosting capabilities to master point-to-point connections with another USB peripheral, initiating transfers on the bus for the peripheral to respond. USB IN packets are received into the RX FIFOs to be moved into the processor core memory, and data written into TX FIFOs is transmitted onto the bus as USB OUT packets. In this mode, the USB controller encodes, decodes, and checks USB packets sent and received. The controller automatically schedules isochronous and interrupt transfers from the endpoint buffers such that one transaction is performed every *n* frames, where *n* represents the polling interval programmed for the endpoint.

Figure 26-1 shows the main functional blocks within the USB controller and its interfaces to the processor core, USB controller RAM, and USB OTG PHY.

Any of the endpoints can be programmed to be written to or read from using the DMA master channels to provide the most efficient means of transferring data between the controller and on-chip memory. USB endpoints 0 through 7 have DMA interrupt lines (`USB_DMAxINT`) providing a total of eight DMA request lines. Three top-level maskable interrupts are provided, each of which can be sourced from any or all of transmit endpoint status, receive endpoint status or global USB status. Details of these can be found in "Interrupts" on page 26-8.

The USB controller uses the peripheral bus to access control and status registers and FIFOs from a slave perspective and to transfer data between the USB engine and on-chip memory as a master. The MMR peripheral data bus is 16-bits wide, the DMA DCB/DEB data bus is also 16-bits wide. Using the 16-bit wide data bus, the USB controller to processor core interface translates into either half word transfers (for both CSR and FIFO addresses) or byte transfers (FIFO addresses only).

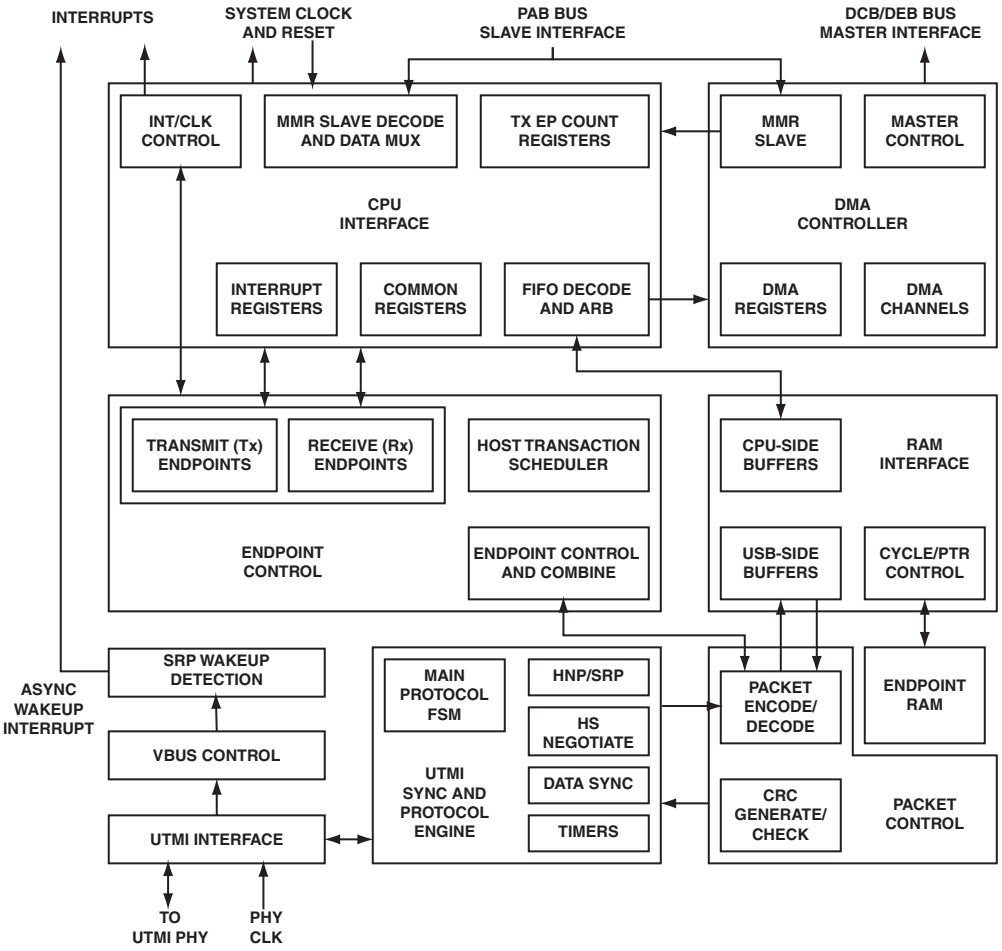


Figure 26-1. USB OTG Controller Block Diagram

The USB controller's RAM interface supports a single block of synchronous single-port RAM used to buffer the USB packets. 7.232K bytes of SRAM are available.

The UTMI+ level 2 PHY interface provides a means of connecting a selection of high- or full-speed PHYs to the controller, from device-only PHYs

through full OTG compliant PHYs. The details of the PHY interface can be found in "UTMI Interface" on page 26-54.

The USB controller requires a system clock frequency of greater than 30 MHz to operate correctly on USB.

The USB controller must not be used if the system clock is operating at a clock frequency below 30 MHz.

The USB controller is configured as either a USB OTG 'A' device or 'B' device depending on the type of plug inserted into its USB receptacle. This is determined by the state of the `USB_ID` (connector ID) pin.

The asynchronous wakeup circuit is used to detect when another 'B' device is asserting its D+ pull-up to initiate the SRP (session request protocol) when all other clocks are off. This circuit requires a slow clock (for example, 32kHz).

Before any endpoint register writes can be committed on endpoint zero, or before control transfers take place, the `GLOBAL_ENA` bit of the `USB_GLOBAL_CTL` register must be set in order to enable the system clock for the control logic. Likewise, before any endpoints can be set up and used to transfer data, the related control bit in the `USB_GLOBAL_CTL` register must be set.

Use of the controller for OTG functionality requires the capability to drive VBUS (as a default 'A' device powering the bus), to discharge VBUS (speeding up the time for VBUS to fall below the `SessionEnd` threshold as a 'B' device checking initial conditions), and to charge VBUS to 2.1V (when initiating SRP as a 'B' device). These controls are driven from the UTMI interface, but the controller also provides a separate interrupt register, `USB_OTG_VBUS_IRQ`, which represents the drive VBUS, discharge VBUS, and charge VBUS signaling. See "USB OTG VBUS Interrupt (USB_OTG_VBUS_IRQ) Register" on page 26-134 for more information on these controls.

FIFO Configuration

Each bidirectional endpoint (provided as two unidirectional endpoints) has its own endpoint number (0 for control, 1–7 for data transfer). Although two endpoints might use the same number, the endpoints may support different transfer types. Each of these bidirectional endpoints has a fixed region of the SRAM in the USB controller to which it has access, and this feature dictates to some extent the types of transfers that may be used for that particular endpoint. This restriction follows from the maximum size of USB packets, which varies with each transfer type. Table 26-1 lists the endpoint FIFO configuration, with an indication of the transfer types possible for that particular buffer size.

Table 26-1. FIFO Sizes and Transfer Types

Bidirectional Endpoint (RX and TX)	FIFO Size (each direction)	USB Transfer Types
0	64 bytes	Size fixed for Control transfers.
1–4	128 bytes	Bulk, Interrupt, Isochronous
5–7	1024 bytes	Bulk, Interrupt, Isochronous

This configuration gives a total USB controller RAM size of 7232 bytes.

Each endpoint FIFO can buffer one or two packets (in double-buffered mode). The double buffered mode is automatically enabled when the software programs a maximum packet size for an endpoint that is equal to or less than half the actual FIFO size for that endpoint. Double-buffering is recommended for most applications to improve efficiency by reducing the frequency with which each endpoint needs to be serviced. Double-buffering Bulk transactions means that data transfer over the USB is not slowed if packets can be loaded/unloaded from the FIFO in the time it takes to transfer a packet over the bus. Double-buffering Isochronous transactions also allows more time to load/unload the FIFO, but in addition, it also

allows the SOF interrupt to be used to service the endpoint rather than the endpoint interrupt. This has the following advantages:

- easy detection of lost packets
- regular interrupt timing (making it easier to source/sink the data)
- If more than one Isochronous endpoint is used, they can all be serviced with one interrupt.

Interrupts

Three active-high top-level interrupts are provided from the USB controller: USB_INT0, USB_INT1 and USB_INT2. Each of these interrupts can be routed through the programming of a global mask register (USB_GLOBINTR) and can be sourced from control transfers, transmit (USB_INTRTX), and receive (USB_INTRRX) endpoint activity, from a range of conditions on the USB lines (USB_INTRUSB), or from requests for the USB controller to send VBUS control signals to an external analog chip (USB_OTG_VBUS_IRQ). The USB_INTRUSB and USB_OTG_VBUS_IRQ sources share the same interrupt line and can not be routed separately (for example, USB_INTRTX and USB_INTRRX). Finally, the DMA master channels use a separate interrupt, USB_DMAxINT, to indicate when a master transfer is pending.

Figure 26-2 shows the various sources of interrupts in the USB controller and how they are routed to the top-level interrupts using the USB_GLOBINTR register.

Interrupts can be generated from control endpoint zero under the following conditions:

- When a control transaction ends before the end of the data is transferred.
- When a data packet is sent or received from the endpoint 0 FIFOs.

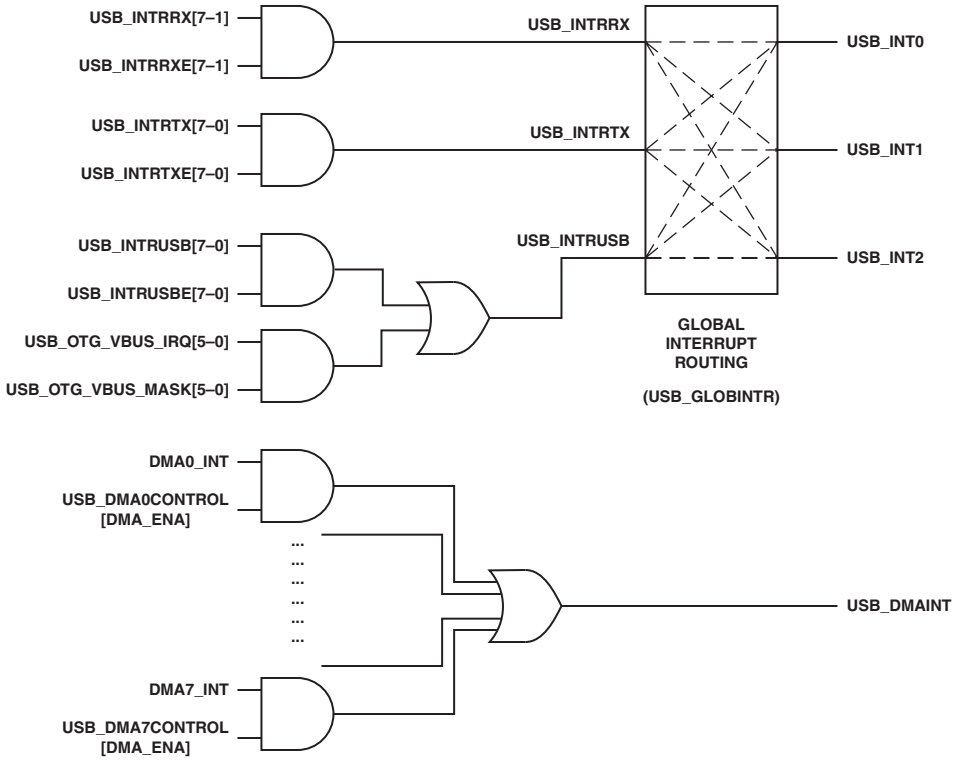


Figure 26-2. USB Interrupt Sources and Routing

Interrupts can be generated from transmit endpoints (`USB_INTRTX`) under the following conditions:

- packet sent from the TX FIFO (host and peripheral mode)
- after three attempts at transmitting a packet with no valid handshake packet received (host mode)

Interrupts can be generated from receive endpoints (USB_INTRRX) under the following conditions:

- packet received into the RX FIFO (host and peripheral mode)
- when a STALL handshake is received (host mode)
- After three attempts at receiving a packet and no data packet is received (host mode).

Interrupts can be generated from the USB status (USB_INTRUSB) under the following conditions:

- When VBUS drops below the VBUS valid threshold during a session ('A' device only).
- When SRP signalling is detected ('A' device only).
- When device disconnect is detected (host mode).
- When a session ends (peripheral mode).
- Device connection detected (host mode).
- Start-of-frame (SOF)
- Reset signalling detected on USB (peripheral mode).
- Babble detected (host mode).
- In suspend mode when resume signalling detected on USB.
- When suspend signalling is detected (peripheral mode).

Interrupts are generated for the following VBUS control requests by the USB controller:

- drive VBUS greater than 4.4V (Default 'A' device)
- stop driving VBUS

- start charging VBUS (peripheral mode)
- stop charging VBUS
- start discharging VBUS (peripheral mode)
- stop discharging VBUS

Resets

The USB controller includes an active-high synchronous hardware reset sourced from the processor core. Another source of peripheral reset is through the USB, when USB reset signaling is detected on the I/O lines. As dictated by the USB 2.0 Specification, this state is entered when both the D+ and D– inputs are driven low for a period of 2.5µs or more (though the reset itself is held for typically greater than 10ms by the USB host).

When a USB reset is detected, the USB controller performs the following actions:

- `USB_FADDR` register set to zero
- `USB_INDEX` register set to zero
- all endpoint FIFOs flushed
- all control and status registers cleared
- all interrupts enabled
- reset interrupt generated

The `USB_INTRUSB`, `USB_OTG_VBUS_IRQ`, `USB_GLOBINTR`, and `USB_GLOBAL_CTL` registers are *not* affected by the USB controller reset. These registers are only reset (along with those listed above) during a system reset.

Description of Operation

The USB OTG interface may operate in peripheral mode or host mode.

When the USB controller is operating in peripheral mode, the controller may be attached to a conventional host (such as a personal computer) or another OTG device operating in host mode. The second device can be high-speed or full-speed. When linked to another peripheral device, the USB controller can also act as the host, and if the other device is also a dual role controller, the two devices can switch roles as required.

The role taken by the USB controller depends on the way the devices are cabled together. Each USB cable has an 'A' and a 'B' device end. If the 'A' end of the cable is plugged into the device containing the USB controller, the USB controller takes the role of the host device and goes into host mode (in this case the `HOST_MODE` bit is set to 1). If the 'B' of the cable is plugged in, the USB controller goes instead into peripheral mode (and the `HOST_MODE` bit remains at 0).

When both devices contain dual role controllers, signaling may be used to switch the roles of the two devices, without switching the cable connecting the two devices. The conditions under which the USB controller may switch between peripheral and host mode are detailed in "Host Negotiation/Configuration" on page 26-80.

Peripheral Mode Operation

USB OTG interface operations for the peripheral mode differ from host mode in a number of ways. The following sections describe peripheral mode operations.

Endpoint Setup

In peripheral mode, there are a few endpoint-specific configuration bits that are used when setting up an endpoint for transfer for all types of

peripheral transfer. They determine how the processor core interacts with the endpoint FIFO.

One key parameter required before transfer can occur through an endpoint is the maximum USB packet size that the endpoint can support. This value is set by the software and depends on a variety of system constraints. These include the size of hardware FIFO available and system latencies as well as the USB transfer type and class being used. The USB_TX_MAX_PACKET or USB_RX_MAX_PACKET defines the maximum amount of data that can be transferred to the selected endpoint in a single frame, and the value must match the programmed maximum individual packet size (*MaxPktSize*) of the standard endpoint descriptor for the endpoint. For TX endpoints, the maximum packet size is programmed using the USB_TX_MAX_PACKET. For RX endpoints, the USB_RX_MAX_PACKET register is used. The maximum packet size must not exceed the actual hardware endpoint FIFO size (see Table 26-1 on page 26-7). Because the USB controller uses a 16-bit interface, the value chosen for *MaxPktSize* should be an even number, as this selection simplifies transferring data between FIFOs and processor core.

If the size of the endpoint FIFO being used is at least twice the USB_RX_MAX_PACKET or USB_TX_MAX_PACKET, double buffering is automatically enabled for that endpoint.

Additional setup parameters are configured using the USB_RXCSR or USB_TXCSR register (depending on whether the endpoint in question is RX or TX). The DMA_ENA bit in this register is used to enable the assertion of the appropriate DMA request whenever the endpoint is able to receive or transmit another packet. The AUTOCLEAR_R and AUTOSET_R/T bits can be used to automatically set the FIFO ready triggers (RXPKTRDY and TXPKTRDY) whenever a packet is transferred to streamline DMA operation for transfers that span multiple packets. Refer to the descriptions in "USB OTG Registers" on page 26-95 for more details on the endpoint control and status registers.

IN Transactions as a Peripheral

When the USB controller is operating in peripheral mode, data for IN transactions is handled through the TX FIFOs. The maximum size of data packet that may be placed in a TX endpoint's FIFO for transmission is programmable and (where applicable) is determined by the value written to the USB_TX_MAX_PACKET register for that endpoint (maximum payload multiplied by the number of transactions per micro-frame).

The maximum packet size set for any endpoint must not exceed the FIFO size (see Table 26-1 on page 26-7).

The USB_TX_MAX_PACKET register should not be written-to while there is data in the FIFO, as unexpected results may occur.

If the size of the TX endpoint FIFO is less than twice the maximum packet size for this endpoint (as set in the USB_TX_MAX_PACKET register), only one packet can be buffered in the FIFO and *single packet buffering* is enabled. As each packet to be sent is loaded into the TX FIFO, the TXPKTRDY bit in USB_TXCSR needs to be set. If the AUTOSET_T bit in USB_TXCSR is set, the TXPKTRDY bit is automatically set when a maximum-sized packet is loaded into the FIFO. For packet sizes less than the maximum, TXPKTRDY always has to be set manually (for example, set by the processor core).

When the TXPKTRDY bit is set, either manually or automatically, the FIFO_NOT_EMPTY_T bit in USB_TXCSR is also set and the packet is ready to be sent. When the packet is successfully sent, both TXPKTRDY and FIFO_NOT_EMPTY_T are cleared and the appropriate TX endpoint interrupt is generated (if enabled). The next packet can then be loaded into the FIFO.

If the size of the TX endpoint FIFO is at least twice the maximum packet size for this endpoint (as set in the USB_TX_MAX_PACKET), two packets can be buffered in the FIFO and *double packet buffering* is enabled. As each packet to be sent is loaded into the TX FIFO, the TXPKTRDY bit in USB_TXCSR needs to be set. If the AUTOSET_T bit in USB_TXCSR is set, the

TXPKTRDY bit automatically is set when a maximum-sized packet is loaded into the FIFO. For packet sizes less than the maximum, TXPKTRDY always has to be set manually (for example, set by the processor core). When the TXPKTRDY bit is set, either manually or automatically, the FIFO_NOT_EMPTY_T bit in USB_TXCSR also is set. TXPKTRDY is then immediately cleared (and an interrupt generated, if enabled). A second packet can now be loaded into the TX FIFO and TXPKTRDY set again (either manually or automatically if the packet is the maximum size). Both packets are now ready to be sent.

When the first packet is successfully sent, TXPKTRDY is cleared and the appropriate TX endpoint interrupt is generated (if enabled) to signal that another packet can now be loaded into the TX FIFO. The state of the FIFO_NOT_EMPTY_T bit at this point indicates how many packets may be loaded. If the FIFO_NOT_EMPTY_T bit is set then there is another packet in the FIFO and only one more packet can be loaded. If the FIFO_NOT_EMPTY_T bit is cleared then there are no packets in the FIFO and two more packets can be loaded.

OUT Transactions as a Peripheral

When the USB controller is operating in peripheral mode, data for OUT transactions is handled through the USB controller's RX FIFOs.

The maximum amount of data received by an RX endpoint in any frame or micro-frame (in high-speed mode) is programmable and is determined by the value written to the USB_EP_NIx_RXMAXP register for that endpoint. This is the maximum payload multiplied by the number of transactions per micro-frame (where applicable). The maximum packet size must not exceed the FIFO size (see Table 26-1 on page 26-7).

If the size of the RX endpoint FIFO is less than twice the maximum packet size for this endpoint (as set in the USB_RX_MAX_PACKET register), only one data packet can be buffered in the FIFO and single packet buffering is enabled. When a packet is received and placed in the RX FIFO, the RXPKTRDY bit and the FIFO_FULL_R bit in USB_RXCSR are set and the

appropriate RX endpoint interrupt is generated (if enabled) to signal that a packet can now be unloaded from the FIFO. After the packet is unloaded, the RXPKTRDY bit needs to be cleared in order to allow further packets to be received. If the AUTOCLEAR_R bit in USB_RXCSR is set and a maximum-sized packet is unloaded from the FIFO, the RXPKTRDY bit is cleared automatically. The FIFO_FULL_R bit is also cleared. For packet sizes less than the maximum, RXPKTRDY always has to be cleared manually (for example, set by the processor core).

If the size of the RX endpoint FIFO is at least twice the maximum packet size for the endpoint, two data packets can be buffered and *double packet buffering* is enabled. When the first packet to be received is loaded into the RX FIFO, the RXPKTRDY bit in USB_RXCSR is set and the appropriate RX endpoint interrupt is generated (if enabled) to signal that a packet can now be unloaded from the FIFO. Note that the FIFO_FULL_R bit in USB_RXCSR is not set at this point. This bit is only set if a second packet is received and loaded into the RX FIFO.

After the first packet is unloaded, RXPKTRDY needs to be cleared in order to allow further packets to be received. If the AUTOCLEAR_R bit in USB_RXCSR is set and a maximum-sized packet is unloaded from the FIFO, the RXPKTRDY bit is cleared automatically. For packet sizes less than the maximum, RXPKTRDY always has to be cleared manually (for example, set by the processor core).

If the FIFO_FULL_R bit was set to 1 when RXPKTRDY is cleared, the USB controller first clears the FIFO_FULL_R bit. The controller then sets RXPKTRDY again to indicate that there is another packet waiting in the FIFO to be unloaded.

Peripheral Transfer Workflows

The USB transfer types (control, bulk, isochronous and interrupt transfers) each have significantly different system requirements as well as individual USB transfer-specific features. This dictates that they are each dealt with slightly differently in software. For these reasons, there is no

uniform way of doing transfers across all transfer types on the USB controller.

The following section provides some guideline peripheral mode transfer flows for each of the transfer types, in both IN (TX) and OUT (RX) directions. In the case of bulk endpoints, the optimal transfer flow differs depending on whether the final size of the transfer is known or unknown. Whether the transfer size is known or not depends on the USB driver class being used. Some define the complete transfer size, and others operate on a packet-by-packet basis using a short packet (a packet of less than `USB_TX_MAX_PACKET` or less than `USB_RX_MAX_PACKET`) to denote the end of a transfer.

Each of the workflows use the following common method.

1. Configure the endpoint control and status registers and the `USB_TX_MAX_PACKET` or `USB_RX_MAX_PACKET` value.
2. Configure the appropriate data transfer mechanism (DMA or interrupt setup).
3. Data transfer phase

The workflows do not describe the USB controller's actions immediately preceding the endpoint setup (for example, the reception of an IN/OUT token from the host, token validity checking, or NAK generation, among others). Note also that there is currently no error-handling contained in the workflows (for example, checking `FIFO_FULL_R` bit before writing data).

The terms packets, frames and transfers are used in the proceeding sections with their strict USB definitions. (See the "Glossary of USB Terms " on page 26-148 for these definitions.)

Control Transactions as a Peripheral

Endpoint 0 is the main control endpoint of the USB controller. As such, the routines required to service Endpoint 0 are more complicated than those required to service other endpoints.

The software is required to handle all the Standard Device Requests that may be sent or received through Endpoint 0. These are described in *Universal Serial Bus Specification*, Revision 2.0, Chapter 9. The protocol for these device requests involves different numbers and types of transactions per transfer. To accommodate this, the processor needs to take a state machine approach to command decoding and handling.

The Standard Device Requests received by a USB peripheral can be divided into three categories: Zero Data Requests (in which all the information is included in the command), Write Requests (in which the command will be followed by additional data), and Read Requests (in which the device is required to send data back to the host).

This section looks at the sequence of actions that the software must perform to process these different types of device request.

The Setup packet associated with a Standard Device Request should include an 8-byte command. A Setup packet containing a command field of anything other than 8 bytes will be automatically rejected by the USB controller.

Write Requests

Write requests involve an additional packet (or packets) of data being sent from the host after the 8-byte command. An example of a 'Write' Standard Device Request is: `SET_DESCRIPTOR`.

The sequence of events will begin, as with all requests, when the software receives an Endpoint 0 interrupt. The `RxPktRdy` bit will also have been set. The 8-byte command should then be read from the Endpoint 0 FIFO and decoded.

As with a zero data request, the USB_CSR0 register should then be written to set the ServicedRxPktRdy bit (indicating that the command is read from the FIFO) but in this case the DataEnd bit should not be set (indicating that more data is expected).

When a second Endpoint 0 interrupt is received, the USB_CSR0 register is read to check the endpoint status. The RxPktRdy bit is set to indicate that a data packet is received. The USB_COUNT0 register should then be read to determine the size of this data packet. The data packet can then be read from the Endpoint 0 FIFO.

If the length of the data associated with the request (indicated by the wLength field in the command) is greater than the maximum packet size for Endpoint 0, further data packets will be sent. In this case, USB_CSR0 is written to set the ServicedRxPktRdy bit, but the DataEnd bit should not be set.

When all the expected data packets have been received, the USB_CSR0 register is written to set the ServicedRxPktRdy bit and to set the DataEnd bit (indicating that no more data is expected).

When the host moves to the status stage of the request, another Endpoint 0 interrupt will be generated to indicate that the request has completed. No further action is required from the software—the interrupt is just a confirmation that the request completed successfully.

If the command is an unrecognized command, or for some other reason cannot be executed, then when it has been decoded, the USB_CSR0 register should be written to set the ServicedRxPktRdy bit and to set the SendStall bit. When the host sends more data, the USB controller will send a STALL to tell the host that the request was not executed. An Endpoint 0 interrupt will be generated and the SentStall bit will be set.

If the host sends more data after the DataEnd has been set, then the USB controller will send a STALL. An Endpoint 0 interrupt will be generated and the SentStall bit will be set.

Read Requests

Read requests have a packet (or packets) of data sent from the function to the host after the 8-byte command. Examples of Standard Device Requests for Read are: GET_CONFIGURATION, GET_INTERFACE, GET_DESCRIPTOR, GET_STATUS, SYNCH_FRAME.

The sequence of events will begin, as with all requests, when the software receives an Endpoint 0 interrupt. The RxPktRdy bit in USB_CSR0 will also have been set. The 8-byte command should then be read from the Endpoint 0 FIFO and decoded. The USB_CSR0 register should then be written to set the ServicedRxPktRdy bit (indicating that the command has read from the FIFO).

The data to be sent to the host should then be written to the Endpoint 0 FIFO. If the data to be sent is greater than the maximum packet size for Endpoint 0, only the maximum packet size should be written to the FIFO. The USB_CSR0 register should then be written to set the TxPktRdy bit (indicating that there is a packet in the FIFO to be sent). When the packet has been sent to the host, another Endpoint 0 interrupt will be generated and the next data packet can be written to the FIFO.

When the last data packet has been written to the FIFO, the USB_CSR0 register should be written to set the TxPktRdy bit and to set the DataEnd bit (indicating that there is no more data after this packet).

When the host moves to the status stage of the request, another Endpoint 0 interrupt will be generated to indicate that the request has completed. No further action is required from the software—the interrupt is just a confirmation that the request completed successfully.

If the command is an unrecognized command, or for some other reason cannot be executed, then when it has been decoded, the USB_CSR0 register should be written to set the ServicedRxPktRdy bit and to set the SendStall bit. When the host requests data, the USB controller will send a STALL to tell the host that the request was not executed. An Endpoint 0 interrupt will be generated and the SentStall bit will be set.

If the host requests more data after DataEnd has been set, then the USB controller will send a STALL. An Endpoint 0 interrupt will be generated and the SentStall bit will be set.

Zero Data Requests

Zero data requests have all their information included in the 8-byte command and require no additional data to be transferred.

Examples of zero data Standard Device Requests are: SET_FEATURE, CLEAR_FEATURE, SET_ADDRESS, SET_CONFIGURATION, SET_INTERFACE.

The sequence of events will begin, as with all requests, when the software receives an Endpoint 0 interrupt. The RxPktRdy bit will also have been set. The 8-byte command should then be read from the Endpoint 0 FIFO, decoded and the appropriate action taken. For example if the command is SET_ADDRESS, the 7-bit address value contained in the command is written to the USB_FADDR register.

When the host moves to the status stage it still addresses the device with the default address, therefore the USB_FADDR should not be written before the host moves to the status stage. In the next transaction the host will then use this new address to address the device.

The USB_CSR0 register should then be written to set the ServicedRxPktRdy bit (indicating that the command is read from the FIFO) and to set the DataEnd bit (indicating that no further data is expected for this request).

When the host moves to the status stage of the request, a second Endpoint 0 interrupt will be generated to indicate that the request has completed. No further action is required from the software—the second interrupt is just a confirmation that the request completed successfully.

If the command is an unrecognized command, or for some other reason cannot be executed, then when it is decoded, the USB_CSR0 register is written to set the ServicedRxPktRdy bit and to set the SendStall bit. When the host moves to the status stage of the request, the USB controller will

send a STALL to tell the host that the request was not executed. A second Endpoint 0 interrupt will be generated and the SentStall bit will be set.

If the host sends more data after the DataEnd bit is set, then the USB controller will send a STALL. An Endpoint 0 interrupt will be generated and the SentStall bit will be set.

ENDPOINT 0 States

When the USB is operating as a peripheral, the Endpoint 0 control needs three modes (IDLE, TX and RX) corresponding to the different phases of the control transfer and the states Endpoint 0 enters for the different phases of the transfer (see "Endpoint 0 Service Routine as Peripheral" on page 26-24).

The default mode on power-up or reset should be IDLE. The RxPktRdy bit becoming set when Endpoint 0 is in IDLE state indicates a new device request. Once the device request is unloaded from the FIFO, the USB decodes the descriptor to find whether there is a data phase and, if so, the direction of the data phase of the control transfer (in order to set the FIFO direction).

Depending on the direction of the data phase, Endpoint 0 goes into either TX state or RX state. If there is no data phase, Endpoint 0 remains in IDLE state to accept the next device request.

The processor needs to take different actions at the different phases of the possible transfers (for example, "Loading the FIFO", "Setting TxPktRdy") are indicated in Figure 26-3 on page 26-23. Note that the USB changes

the FIFO direction depending on the direction of the data phase, independently of the processor.

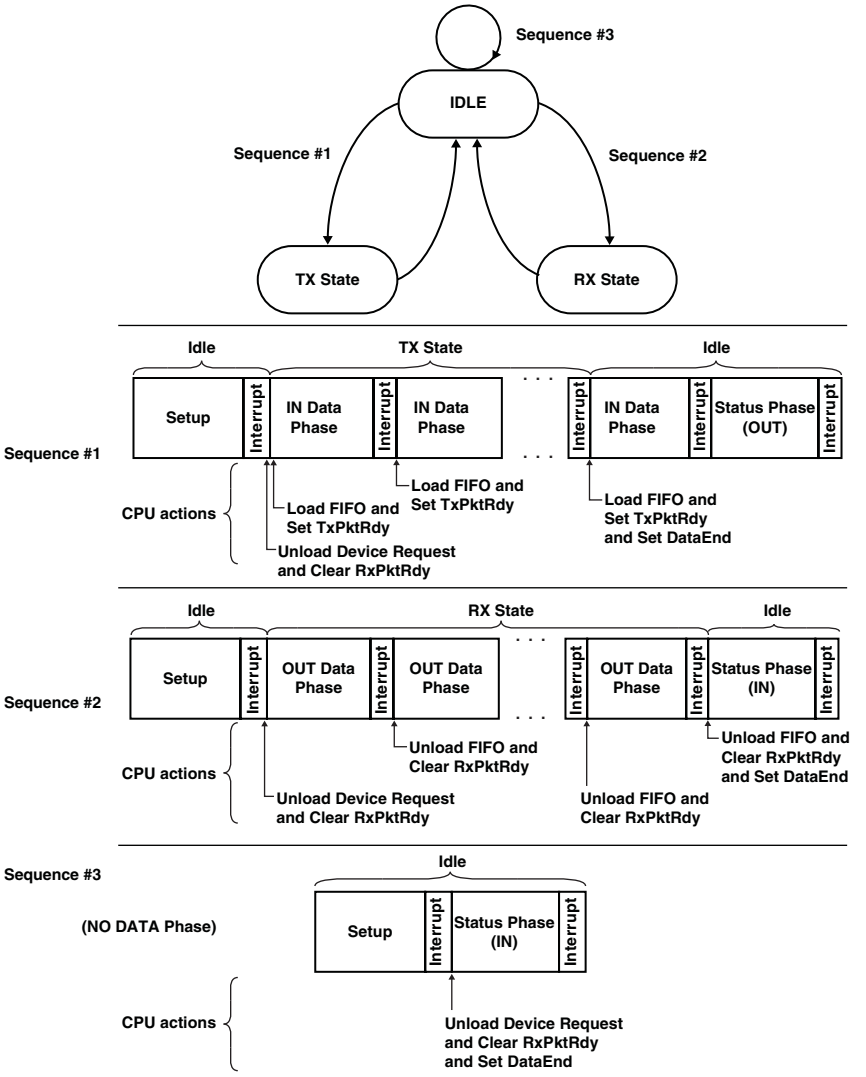


Figure 26-3. Endpoint 0 Control States

Endpoint 0 Service Routine as Peripheral

An Endpoint 0 interrupt is generated:

- When the USB controller sets the `RxPktRdy` bit after a valid token has been received and data has been written to the FIFO.
- When the USB controller clears the `TxPktRdy` bit after the data packet in the FIFO has been successfully transmitted to the host.
- When the USB controller sets the `SentStall` bit after a control transaction is ended due to a protocol violation.
- When the USB controller sets the `SetupEnd` bit because a control transfer has ended before `DataEnd` is set.

The bits mentioned above, are in the `USB_CSR0` register.

Whenever the Endpoint 0 service routine is entered, the firmware must first check whether the current control transfer has been ended due to either a `STALL` condition or a premature end-of-control transfer. If the control transfer ends due to a `STALL` condition, the `SentStall` bit would be set. If the control transfer ends due to a premature end-of-control transfer, the `SetupEnd` bit would be set. In either case, the firmware should abort processing the current control transfer and set the state to IDLE.

Once the firmware has determined that the interrupt was not generated by an illegal bus state, the next action depends on the Endpoint state.

If Endpoint 0 is in IDLE state, the only valid reason an interrupt can be generated is as a result of the core receiving data from the USB bus. The service routine must check for this by testing the `RxPktRdy` bit. If this bit is set, then the core has received a SETUP packet. This must be unloaded from the FIFO and decoded to determine the action the core must take.

Depending on the command contained within the SETUP packet, Endpoint 0 will enter one of three states:

- If the command is a single packet transaction (`SET_ADDRESS`, `SET_INTERFACE` etc.) without a data phase, the endpoint will remain in IDLE state.
- If the command has an OUT data phase (`SET_DESCRIPTOR` etc.), the endpoint will enter RX state.
- If the command has an IN data phase (`GET_DESCRIPTOR` etc.), the endpoint will enter TX state.

If the endpoint is in TX state, the interrupt indicates that the core has received an IN token and data from the FIFO has been sent.The firmware must respond to this either by placing more data in the FIFO if the host is still expecting more data[1] or by setting the `DataEnd` bit to indicate that the data phase is complete. Once the data phase of the transaction has been completed, Endpoint 0 should be returned to IDLE state to await the next control transaction.

If the endpoint is in RX state, the interrupt indicates that a data packet has been received. The firmware must respond by unloading the received data from the FIFO. The firmware must then determine whether it has received all of the expected data[1]. If it has, the firmware should set the `DataEnd` bit and return Endpoint 0 to IDLE state. If more data is expected, the firmware should set the `ServicedRxPktRdy` bit to indicate that it has read the data in the FIFO and leave the endpoint in RX state.

[1] Command transactions all include a field that indicates the amount of data the host expects to receive or is going to send.

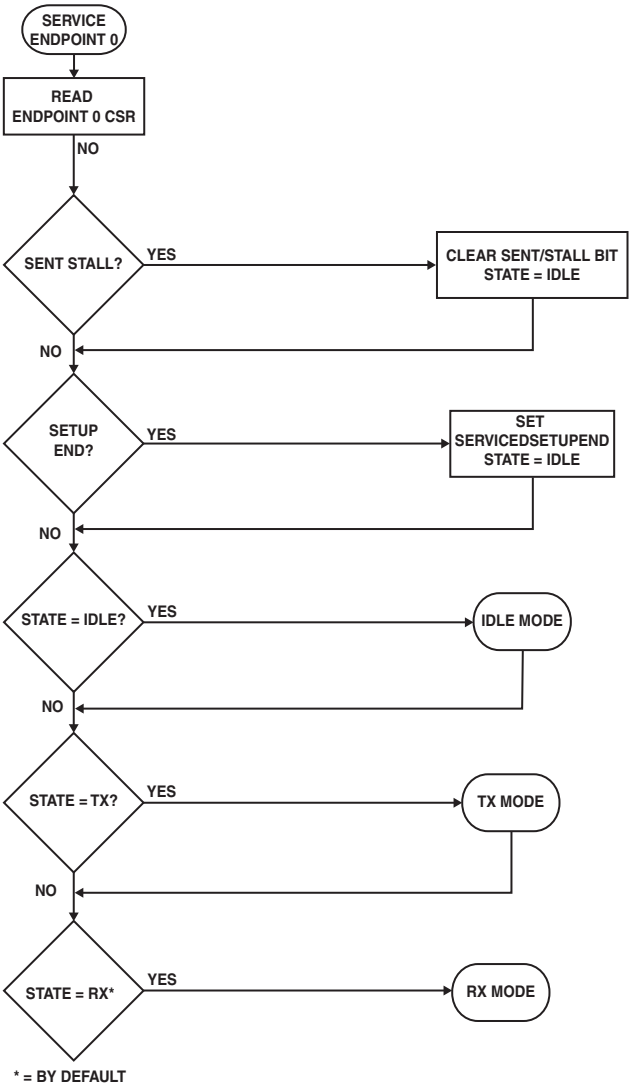


Figure 26-4. Endpoint 0 Service Routine

Idle Mode

The Endpoint 0 control must select the IDLE mode at power-on or reset. The Endpoint 0 control should return to this mode when the RX and TX modes are terminated.

This is also the mode in which the SETUP phase of control transfer is handled (see Figure 26-5 on page 26-28).

TX Mode

When the endpoint is in TX state, all arriving IN tokens need to be treated as part of a data phase until the required amount of data has been sent to the host. If either a SETUP or an OUT token is received while the endpoint is in the TX state, a `SetupEnd` condition would occur since the core expects only IN tokens.

Three events can cause the TX mode to terminate before the expected amount of data has been sent:

- The host sends an invalid token causing a `SetupEnd` bit set.
- The firmware sends a packet containing less than the maximum packet size for Endpoint 0.
- The firmware sends an empty data packet.

Until the transaction is terminated, when the firmware receives an interrupt which indicates that a packet has been sent from the FIFO, it simply loads the FIFO. An interrupt is generated when TxPktRdy is cleared.

When the firmware forces the termination of a transfer (by sending a short or empty data packet), it should set the `DataEnd` bit to indicate to the core

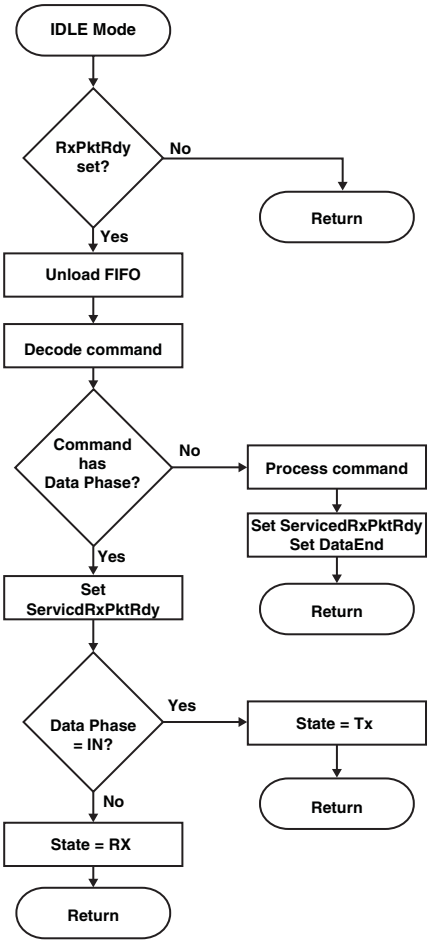


Figure 26-5. Endpoint 0 Idle Mode (Setup Phase)

that the data phase is complete and that the core should receive an acknowledge packet next.

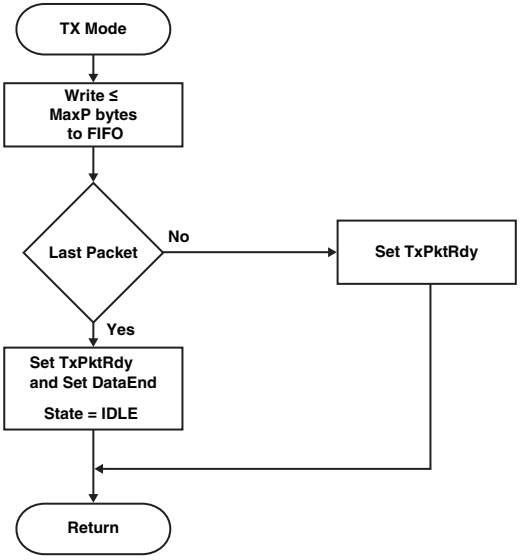


Figure 26-6. Endpoint 0 TX Mode

RX Mode

In RX mode, all arriving data should be treated as part of a data phase until the expected amount of data has been received. If either a SETUP or an IN token is received while the endpoint is in RX state, a `SetupEnd` condition would occur since the core expects only OUT tokens.

Three events can cause the RX mode to terminate before the expected amount of data has been received:

- The host sends an invalid token causing a `SetupEnd` bit set.
- The host sends a packet which contains less than the maximum packet size for Endpoint 0.
- The host sends an empty data packet.

Until the transaction is terminated, when the firmware receives an interrupt which indicates that new data has arrived (`RxPktRdy` bit set), it simply needs to unload the FIFO and clear `RxPktRdy` by setting the `ServicedRxPktRdy` bit.

When the firmware detects the termination of a transfer (by receiving either the expected amount of data or an empty data packet), it should set

the DataEnd bit to indicate to the core that the data phase is complete and that the core should receive an acknowledge packet next.

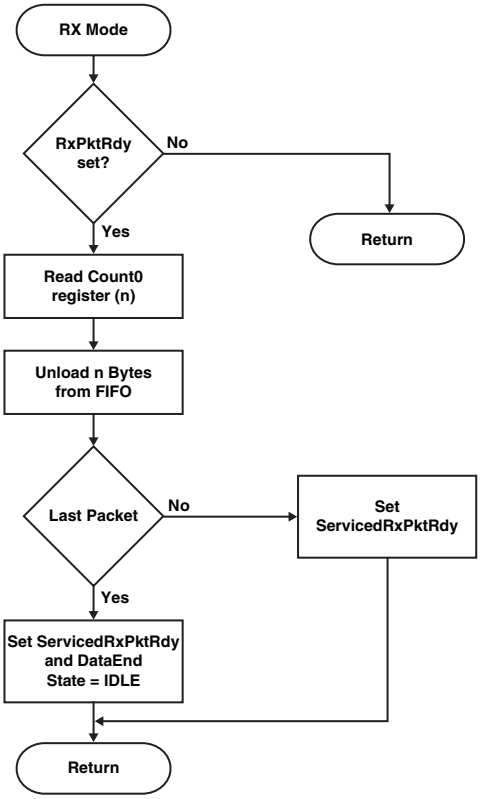


Figure 26-7. Endpoint 0 RX Mode

If the last packet is not a multiple of four bytes it is strongly recommended that the remainder (n bytes mod 4) be unloaded from the FIFO using the byte addressing FIFO register (EP0 FIFO address + 4). This will prevent the USB controller from sending non-null data during the status phase of the control transfer.

Peripheral Mode, Bulk IN, Transfer Size Known

For this process, the maximum individual packet size (*MaxPktSize*) in bytes and the complete transfer size (*TxferSize*) in bytes, must be known.

1. Load *MaxPktSize* into USB_TX_MAX_PACKET.
2. Set DMA_ENA = 1, AUTOSET_T = 1, ISO_T = 0, FRCDATATOG = 0 in USB_TXCSR.
3. Load *TxferSize* into USB_TXCOUNT.
4. Configure the DMA controller to write full *TxferSize*/2 half words into the corresponding TX FIFO address.
5. On each USB_DMAxINT transition, the DMA controller writes a new packet into the FIFO. TXPKTRDY is automatically set when each new packet is written.
6. Step 5 is repeated for each full packet of the transfer.
7. Even if the final packet is a short packet, the packet automatically is detected by the USB controller (because USB_TXCOUNT is zero) and TXPKTRDY is set.

Peripheral Mode, Bulk IN, Transfer Size Unknown

For this process, the maximum individual packet size (*MaxPktSize*) in bytes is assumed to be an even number of bytes.

1. Load *MaxPktSize* into USB_TX_MAX_PACKET.
2. Set DMA_ENA = 1, AUTOSET_T = 1, ISO_T = 0, FRCDATATOG = 0 in USB_TXCSR.
3. Configure the DMA controller to write *MaxPktSize*/2 half words into the corresponding TX FIFO address on each USB_DMAxINT.

4. Set up an ISR, sensitive to the DMA work-block-complete interrupt, that writes a remaining short packet into the TX FIFO using processor core DMA. Then set TXPKTRDY or simply send a zero-length packet by toggling TXPKTRDY.

5. On each USB_DMAxINT transition, the DMA controller writes a new packet into the FIFO. TXPKTRDY automatically is set when each new packet is written.

6. Step 5 is repeated for each full packet of the transfer.

7. The final short/zero-length packet is managed by the ISR from step 4.

Peripheral Mode, ISO IN, Small *MaxPktSize*

For this process, the maximum individual packet size (*MaxPktSize*) in bytes is less than 128 bytes and is an even number of bytes. Double buffering is assumed to be enabled, and the auto set feature unused (because packets are often less than *MaxPktSize*).

1. Load *MaxPktSize* into USB_TX_MAX_PACKET.

2. Set ISO_T = 1 in USB_TXCSR.

3. Preload the first two packets into the endpoint TX FIFO and set TXPKTRDY (or alternatively use the USB_TXCOUNT feature that sets TXPKTRDY after USB_TXCOUNT bytes have been loaded).

4. Set up an ISR, sensitive to the SOF_B interrupt, which writes a new packet into the TX FIFO and sets TXPKTRDY.

5. Set SOF_B = 1 in USB_INTRUSBE to generate an interrupt on each start-of-frame.

6. Step 5 is repeated for each ISO packet.

Peripheral Mode, ISO IN, Large *MaxPktSize*

For this process, the maximum individual packet size (*MaxPktSize*) in bytes is greater than 128 bytes and is an even number of bytes. Double buffering is assumed to be enabled, and the auto set feature unused (because packets are often less than *MaxPktSize*).

1. Load *MaxPktSize* into USB_TX_MAX_PACKET.
2. Set ISO_T = 1 in USB_TXCSR.
3. Set ISO_UPDATE = 1 in USB_POWER to prevent initial packet loaded into the FIFO from being transmitted on USB until the next 1ms frame.
4. Load the total number of bytes for the first two packets into USB_TXCOUNT.
5. Configure the DMA controller to pre-load the two packets (as half words) into the corresponding TX FIFO address. TXPKTRDY automatically is set by the USB controller when USB_TXCOUNT bytes have been loaded.
6. Set up an ISR, sensitive to the SOF_B interrupt, which writes a new packet into the TX FIFO by loading USB_TXCOUNT with the size of the packet, then configuring the DMA controller to load the packet.
7. Set SOF_B = 1 in USB_INTRUSBE to generate an interrupt on each start-of-frame.
8. Step 7 is repeated for each ISO packet.

Peripheral Mode, Bulk OUT, Transfer Size Known

For this process, the maximum individual packet size (*MaxPktSize*) in bytes and the complete transfer size (*TxferSize*) in bytes must be known.

1. Load *MaxPktSize* into USB_RX_MAX_PACKET.
2. Set DMA_ENA = 1, AUTOCLEAR_R = 1, ISO_R = 0, FRCDATATOG = 0, DMAREQMODE_R = 0 in USB_RXCSR.
3. Configure the DMA controller to read the full *TxferSize*/2 half words from the corresponding RX FIFO address.
4. On each USB_DMAxINT transition, the DMA controller reads another packet from the FIFO. RXPKTRDY is automatically cleared by the USB controller when each new packet is read.
5. Step 5 is repeated for each full packet of the transfer.
6. If *TxferSize* is not an exact multiple of *MaxPktSize*, the final USB_DMAxINT transition causes the DMA controller to read out only the short packet that remains.

Peripheral Mode, Bulk OUT, Transfer Size Unknown

For this process, the maximum individual packet size (*MaxPktSize*) in bytes must be known.

1. Load *MaxPktSize* into USB_RX_MAX_PACKET.
2. Set DMA_ENA = 1, AUTOCLEAR_R = 1, ISO_R = 0, FRCDATATOG = 0, DMAREQMODE_R = 1 in USB_RXCSR.
3. Set the appropriate EPx_RX_E bit in USB_INTRRXE.
4. Configure the DMA controller to read *MaxPktSize*/2 half words from the corresponding RX FIFO address on each USB_DMAxINT transition.

5. Set up an ISR, sensitive to the RX interrupt, which reads USB_RXCOUNT and then transfers USB_RXCOUNT bytes (in half words) from the RX FIFO to the processor core. Depending on the number of bytes in the FIFO, this can be performed by configuring the DMA to read the data, or by reading it with the processor core.
6. On each USB_DMAxINT transition, the DMA controller reads a packet from the FIFO. RXPKTRDY is automatically cleared by the USB controller when each new packet is read.
7. Step 5 is repeated for each full packet of the transfer.
8. If a packet is received that is less than *MaxPktSize*, the RX interrupt goes high, and the ISR from step 5 reads out the remaining short packet.

Peripheral Mode, ISO OUT, Small *MaxPktSize*

For this process, the maximum individual packet size (*MaxPktSize*) in bytes is less than 128 bytes, and double buffering is assumed to be enabled.

1. Load *MaxPktSize* into USB_RX_MAX_PACKET.
2. Set ISO_R = 1 in USB_RXCSR.
3. Set up an ISR, sensitive to the SOF_B interrupt, that reads the FIFO_FULL_R bit, reads the USB_RXCOUNT status register, and finally removes one or two packets (equal to the USB_RXCOUNT number of bytes) from the FIFO then clears RXPKTRDY.
4. Set SOF_B = 1 in USB_INTRUSBE to generate an interrupt on each start-of-frame.
5. Step 4 is repeated for each ISO packet.

Peripheral Mode, ISO OUT, Large *MaxPktSize*

For this process, the maximum individual packet size (*MaxPktSize*) in bytes is greater than 128 bytes, and double buffering is assumed to be enabled.

1. Load *MaxPktSize* into USB_RX_MAX_PACKET.
2. Set ISO_R = 1 in USB_RXCSR.
3. Set up an ISR, sensitive to the SOF_B interrupt, that reads the FIFO_FULL_R bit, reads the USB_RXCOUNT status register, and finally configures the DMA controller to remove one or two packets (equal to the USB_RXCOUNT number of bytes) from the FIFO.
4. Set up an ISR, sensitive to the DMA work-block-complete interrupt to clear RXPKTRDY.
5. Set SOF_B = 1 in USB_INTRUSBE to generate an interrupt on each start-of-frame.
6. Step 5 is repeated for each ISO packet.

Peripheral Mode Suspend

When no activity has occurred on the USB for 3 ms, the USB controller enters suspend mode. If the suspend interrupt (SUSPEND_B) is enabled, an interrupt is generated at this time.

When resume signaling is detected, the USB controller exits suspend mode. If the RESUME_B interrupt is enabled, an interrupt is generated. The processor core can also force the USB controller to exit suspend mode by setting the RESUME_MODE bit in the USB_POWER register. When this bit is set, the USB controller exits suspend mode and drives resume signaling onto the bus. The processor core should clear this bit after 10 ms (a maximum of 15 ms) to end resume signaling.

No RESUME_B interrupt is generated when suspend mode is exited by the processor core.

Start-of-frame (SOF) Packets

When the USB controller is operating in peripheral mode, it should receive a start-of-frame packet from the host every millisecond when in full-speed mode, or every 125 microseconds when in high-speed mode.

When the SOF packet is received, the 11-bit frame number contained in the packet is written into the USB_FRAME register and an output pulse, lasting one USB clock bit period, is generated on SOF_PULSE (internal USB controller signal). A SOF_B interrupt is also generated (if enabled in the USB_INTRUSBE register).

After the USB controller has started to receive SOF packets, the controller expects one every millisecond (or 125 µs when in high-speed mode). If no SOF packet is received after 1.00358 ms (or 125.125 µs), it is assumed that the packet is lost. An SOF_PULSE (together with a SOF_B interrupt, if enabled) is still generated though the USB_FRAME register is not updated. The USB controller continues to generate an SOF_PULSE every millisecond (or 125 µs) and re-synchronizes these pulses to the received SOF packets when these packets are successfully received again.

Soft Connect/Soft Disconnect

In peripheral mode, the USB controller can be programmed to switch between normal mode and non-driving mode by setting or clearing the SOFT_CONN bit of the USB_POWER register. When this SOFT_CONN bit is set to 1, the USB controller is placed in its normal mode and the D+/D– lines of the USB bus are enabled. When the SOFT_CONN bit is zero, the PHY is put into non-driving mode and D+ and D– are three-stated. The USB controller appears to have been disconnected from the USB bus.

After system reset, SOFT_CONN is cleared to 0. From that point, the USB controller appears disconnected until the software has set SOFT_CONN to 1.

The application software can then choose when to set the PHY to its normal mode. Systems with a lengthy initialization procedure may use this to ensure that initialization is complete and the system is ready to perform enumeration before connecting to the USB.

Error Handling As a Peripheral

A control transfer may be aborted due to a protocol error on the USB, the host prematurely ending the transfer, or if the function controller software wishes to abort the transfer (for example, because it cannot process the command).

The USB controller will automatically detect protocol errors and send a `STALL` packet to the host under the following conditions:

1. The host sends more data during the OUT data phase of a write request than was specified in the command. This condition is detected when the host sends an OUT token after the `DataEnd` bit is set.
2. The host requests more data during the IN data phase of a read request than was specified in the command. This condition is detected when the host sends an IN token after the `DataEnd` bit in the `USB_CSR0` register is set.
3. The host sends more than *MaxPktSize* data bytes in an OUT data packet.
4. The host sends a non-zero length DATA1 packet during the status phase of a read request.

When the USB controller has sent the `STALL` packet, it sets the `SentStall` bit and generates an interrupt. When the software receives an Endpoint 0 interrupt with the `SentStall` bit set, it should abort the current transfer, clear the `SentStall` bit, and return to the IDLE state.

If the host prematurely ends a transfer by entering the status phase before all the data for the request is transferred, or by sending a new SETUP packet before completing the current transfer, then the SetupEnd bit will be set and an Endpoint 0 interrupt generated. When the software receives an Endpoint 0 interrupt with the SetupEnd bit set, it should abort the current transfer, set the ServicedSetupEnd bit, and return to the IDLE state. If the RxPktRdy bit is set, this indicates that the host has sent another SETUP packet and the software should then process this command.

If the software wants to abort the current transfer, because it cannot process the command or has some other internal error, then it should set the SendStall bit. The USB controller will then send a STALL packet to the host, set the SentStall bit and generate an Endpoint 0 interrupt.

Stalls Issued to Control Transfers

In peripheral mode, the USB controller automatically issues a STALL handshake to a control transfer under the following conditions:

1. The host sends more data during an OUT data phase of a control transfer than was specified in the device request during the SETUP phase. This condition is detected by the USB controller when the host sends an OUT token (instead of an IN token) after the processor core has unloaded the last OUT packet and set DATAEND.
2. The host requests more data during an IN data phase of a control transfer than was specified in the device request during the SETUP phase. This condition is detected by the USB controller when the host sends an IN token (instead of an OUT token) after the processor core has cleared TXPKTRDY and set DATAEND in response to the ACK issued by the host to what should have been the last packet.
3. The host sends more than *MaxPktSize* data with an OUT data token.

4. The host sends the wrong PID (packet identifier) for the OUT status phase of a control transfer.
5. The host sends more than a zero length data packet for the OUT status phase.

Zero Length OUT Data Packets in Control Transfers

A zero-length OUT data packet is used to indicate the end of a control transfer. In normal operation, such packets should only be received after the entire length of the device request is transferred (for example, after the processor core has set DATAEND). If the host sends a zero-length OUT data packet before the entire length of device request is transferred, this packet signals the premature end of the transfer. In this case, the USB controller automatically flushes any IN token loaded by processor core ready for the data phase from the FIFO and sets SETUPEND.

Host Mode Operation

USB OTG interface operations in host mode differ from peripheral mode in a number of ways. The following sections describe host mode operations.

Endpoint Setup and Data Transfer

When the HOST_MODE bit is set to 1, the USB controller operates as a host for point-to-point communications with another USB device. This second device may be either a high-speed, full-speed, or low-speed USB function, but it may not be a hub. Control, bulk, isochronous or interrupt transactions are supported between the USB controller and the second device.

Transfers between the subsystem and endpoint FIFOs in host mode are similar to peripheral mode. With this in mind, see many of the descriptions of processor core to FIFO data transfer in "Peripheral Mode Operation" on page 26-12.

Control Transaction as a Host

Host control transactions are conducted through Endpoint 0. The software is required to handle all the Standard Device Requests that may be sent or received through Endpoint 0 (as described in *Universal Serial Bus Specification*, Revision 2.0, Chapter 9).

For a USB peripheral, there are three categories of Standard Device Requests to be handled: Zero Data Requests (in which all the information is included in the command), Write Requests (in which the command will be followed by additional data), and Read Requests (in which the device is required to send data back to the host).

Zero Data Requests comprise a SETUP command followed by an IN status phase.

Write Requests comprise a SETUP command, followed by an OUT data phase followed by an IN status phase.

Read Requests comprise a SETUP command, followed by an IN data phase followed by an OUT status phase.

A timeout may be set to limit the length of time during which the USB controller will retry a transaction that is continually NAKed by the target. This limit can be between 2 and 2^{15} frames/microframes and is set through the `USB_NAKLIMIT0` register.

The following sections look at the steps in different phases of a control transaction to describe the actions of the core in issuing Standard Device Requests.

Before initiating transactions as a host, the `USB_FADDR` register needs to be set to address the peripheral device. When the device is first connected, `USB_FADDR` is set to zero. After a `SET_ADDRESS` command is issued, `USB_FADDR` is set to the target's new address.

Setup Phase as a Host

The processor core driving the host device performs the following actions for the SETUP phase of a control transaction.

1. Load the eight bytes of the required device request command into the Endpoint 0 FIFO
2. Set SETUPPKT_H (bit 3)and TxPtRdy (bit 1) of the USB_CSR0 register. These bits must be set together.

   The USB controller then sends a SETUP token followed by the 8-byte command to Endpoint 0 of the addressed device, retrying as necessary.
3. At the end of the attempt to send the data, the USB controller will generate an Endpoint 0 interrupt (for example, set IntrTx.D0). The processor core should then read USB_CSR0 to establish whether the STALL_RECEIVED_H bit, the ERROR_H bit or the NAK_TIMEOUT_H bit is set.

   If STALL_RECEIVED_H is set, it indicates that the target did not accept the command (for example, because it is not supported by the target device) and so has issued a STALL response.

   If ERROR_H is set, it means that the USB controller has tried to send the SETUP packet and the following data packet three times without getting a response.

   If NAK_TIMEOUT_H is set, it means that the USB controller has received a NAK response to each attempt to send the SETUP packet, for longer than the time set in the USB_NAKLIMIT0 register. The USB controller can then be directed either to continue trying

this transaction (until it times out again) by clearing the NAK_TIMEOUT_H bit or to abort the transaction by flushing the FIFO before clearing the NAK_TIMEOUT_H bit.

4. If none of STALL_RECEIVED_H, ERROR_H or NAK_TIMEOUT_H is set, the SETUP phase is correctly acknowledged and the processor core should proceed to the following IN data phase, OUT data phase or IN status phase specified for the particular Standard Device Request.

IN Data Phase as a Host

The processor core driving the host device performs the following actions for the IN data phase of a control transaction.

1. Set REQPKT_H in USB_CSR0.
2. Wait while the USB controller sends the IN token and then receives the required data back.
3. When the USB controller generates the Endpoint 0 interrupt (for example, by setting EP0_TX in the USB_INTRTX register)—read USB_CSR0 to establish whether the STALL_RECEIVED_H bit, the ERROR_H bit, the NAK_TIMEOUT_H bit or RxPktRdy is set.

   If STALL_RECEIVED_H is set, it indicates that the target has issued a STALL response.

   If ERROR_H is set, it means that the USB controller has tried to send the required IN token three times without getting a response.

   If NAK_TIMEOUT_H is set, it means that the USB controller has received a NAK response to each attempt to send the IN token, for longer than the time set in the USB_NAKLIMIT0 register. The USB controller can then be directed either to continue trying this trans-

action (until it times out again) by clearing the NAK_TIMEOUT_H bit; or to abort the transaction by clearing REQPKT_H before clearing the NAK_TIMEOUT_H bit.

4. If RxPktRdy is set, the processor core should read the data from the Endpoint 0 FIFO, then clear RxPktRdy.
5. If further data is expected, the processor core should repeat the previous steps.

   When all the data is successfully received, the processor core should proceed to the OUT status phase of the control transaction.

OUT Data as a Host (Control)

The processor core driving the host device performs the following actions for the OUT data phase of a control transaction.

1. Load the data to be sent into the Endpoint 0 FIFO
2. Set the TxPtRdy bit in USB_CSR0.

   The USB controller then proceeds to send an OUT token followed by the data from the FIFO to Endpoint 0 of the addressed device, retrying as necessary.
3. At the end of the attempt to send the data, the USB controller will generate an Endpoint 0 interrupt (for example, set EP0_TX in USB_INTRTX register). The processor core should then read USB_CSR0 to establish whether the STALL_RECEIVED_H bit (D2), the ERROR_H bit (D4) or the NAK_TIMEOUT_H bit (D7) is set.

   If STALL_RECEIVED_H is set, it indicates that the target has issued a STALL response.

   If ERROR_H is set, it means that the USB controller has tried to send the OUT token and the following data packet three times without getting a response.

If NAK_TIMEOUT_H is set, it means that the USB controller has received a NAK response to each attempt to send the OUT token, for longer than the time set in the USB_NAKLIMIT0 register. The USB controller can then be directed either to continue trying this transaction (until it times out again) by clearing the NAK_TIMEOUT_H bit; or to abort the transaction by flushing the FIFO before clearing the NAK_TIMEOUT_H bit.

If none of STALL_RECEIVED_H, ERROR_H or USB_NAKLIMIT0 is set, the OUT data is correctly acknowledged.

4. If further data needs to be sent, the processor core should repeat the previous steps.

   When all the data is successfully sent, the processor core should proceed to the IN status phase of the control transaction.

IN Status Phase as a Host (Following SETUP Phase or OUT Data Phase)

The processor core driving the host device performs the following actions for the IN status phase of a control transaction.

1. Set STATUSPKT_H_H and REQPKT_H (bits 6 and 5 of USB_CSR0, respectively). These bits must be set together.
2. Wait while the USB controller both sends an IN token and receives a response from the USB peripheral.
3. When the USB controller generates the Endpoint 0 interrupt (for example, sets EP0_TX in USB_INTRTX register), read USB_CSR0 to establish whether the STALL_RECEIVED_H bit, the ERROR_H bit, the NAK_TIMEOUT_H bit or RxPktRdy is set.

   If STALL_RECEIVED_H is set, it indicates that the target could not complete the command and so has issued a STALL response.

If ERROR_H is set, it means that the USB controller has tried to send the required IN token three times without getting a response.

If NAK_TIMEOUT_H is set, it means that the USB controller has received a NAK response to each attempt to send the IN token, for longer than the time set in the USB_NAKLIMIT0 register. The USB controller can then be directed either to continue trying this transaction (until it times out again) by clearing the NAK_TIMEOUT_H bit or to abort the transaction by clearing REQPKT_H and STATUSPKT_H_H before clearing the NAK_TIMEOUT_H bit.

4. If RxPktRdy is set, the processor core should simply clear RxPktRdy.

OUT Status Phase as a Host (following IN Data Phase)

The processor core driving the host device performs the following actions for the OUT status phase of a control transaction.

1. Set STATUSPKT_H and TxPktRdy bits. These bits must be set together.
2. Wait while the USB controller both sends the OUT token and a zero-length DATA1 packet.
3. At the end of the attempt to send the data, the USB controller will generate an Endpoint 0 interrupt. The processor core should then read USB_CSR0 to establish whether the STALL_RECEIVED_H bit, the ERROR_H bit or the NAK_TIMEOUT_H bit is set.

   If STALL_RECEIVED_H is set, it indicates that the target could not complete the command and so has issued a STALL response.

   If ERROR_H is set, it means that the USB controller has tried to send the STATUS packet and the following data packet three times without getting a response.

If NAK_TIMEOUT_H is set, it means that the USB controller has received a NAK response to each attempt to send the IN token, for longer than the time set in the USB_NAKLIMITO register. The USB controller can then be directed either to continue trying this transaction (until it times out again) by clearing the NAK_TIMEOUT_H bit or to abort the transaction by flushing the FIFO before clearing the NAK_TIMEOUT_H bit.

4. If none of STALL_RECEIVED_H, ERROR_H or NAK_TIMEOUT_H is set, the status phase is correctly acknowledged.

Host IN Transactions

When the USB controller operates as a host, IN transactions are handled like OUT transactions are handled when the USB controller is operating as a peripheral. But the transaction must first be initiated by setting the REQPKT_H bit in USB_RXCSR. This bit indicates to the transaction scheduler that there is an active transaction on this endpoint. The transaction scheduler then sends an IN token to the target function.

When the packet is received and placed in the RX FIFO, the RXPKTRDY bit in USB_RXCSR is set, and the appropriate RX endpoint interrupt is generated (if enabled) to signal that a packet can now be unloaded from the FIFO. When the packet is unloaded, RXPKTRDY is cleared. The AUTOCLEAR_R bit in the USB_RXCSR register can be used to have RXPKTRDY automatically cleared when a maximum sized packet is unloaded from the FIFO. There is also an AUTOREQ_RH bit in USB_RXCSR that causes the REQPKT_H bit to be automatically set when the RXPKTRDY bit is cleared. The AUTOCLEAR_R and AUTOREQ_RH bits can be used with an external DMA controller to perform complete bulk transfers without processor core intervention.

If the target function responds to a bulk or interrupt IN token with a NAK, the USB controller keeps retrying the transaction until the NAK limit set (in USB_NAKLIMITO) is reached. If the target function responds with a STALL, the USB controller does not retry the transaction, but does

interrupt the processor core with the RXSTALL_TH bit in the USB_RXCSR register set. If the target function does not respond to the IN token within the required time (or there was a CRC or bit-stuff error in the packet), the USB controller retries the transaction. If after three attempts the target function still has not responded, the USB controller clears the REQPKT_H bit and interrupts the processor core with the DATAERROR_R bit in USB_RXCSR set.

Host OUT Transactions

When the USB controller operates as a host, OUT transactions are handled in a similar manner to the way IN transactions are handled when the USB controller operates as a peripheral.

The TXPKTRDY bit in the USB_TXCSR register needs to be set as each packet is loaded into the TX FIFO and the AUTOSET_T bit in USB_TXCSR can be used to cause the TXPKTRDY bit to be automatically set when a maximum sized packet is loaded into the FIFO. The AUTOSET_T bit can be used with an external DMA controller to perform complete bulk transfers without processor core intervention.

If the target function responds to the OUT token with a NAK, the USB controller keeps retrying the transaction until the NAK limit set in USB_NAKLIMITO is reached. If the target function responds with a STALL, the USB controller does not retry the transaction, but does interrupt the processor core with the RXSTALL_TH bit in the USB_TXCSR register set. If the target function does not respond to the OUT token within the required time (or there was a CRC or bit-stuff error in the packet), the USB controller retries the transaction. If after three attempts the target function still has not responded, the USB controller flushes the FIFO and interrupts the processor core with the ERROR_TH bit in USB_TXCSR set.

Transaction Scheduling

When operating as a host, the USB controller maintains a frame counter. If the target function is a full-speed device, the USB controller automati-

cally sends an SOF packet at the start of each frame or micro-frame. If the target function is a low-speed device, a K state is transmitted on the bus to act as a *keep-alive* to stop the low-speed device from going into suspend mode.

After the SOF packet is transmitted, the USB controller cycles through all the endpoints looking for active transactions. An active transaction is defined as an RX endpoint for which the REQPKT_H bit is set or a TX endpoint for which the TXPKTRDY bit is set. An active isochronous or interrupt transaction will only start if it is found on the first transaction scheduler cycle of a frame and if the interval counter for that endpoint has counted down to zero. This ensures that only one interrupt or isochronous transaction occurs per endpoint per *n* frames (where *n* is the interval set in the USB_TXINTERVAL or USB_RXINTERVAL register for that endpoint).

An active bulk transaction is started immediately, provided there is sufficient time left in the frame to complete the transaction before the next SOF packet is due. If the transaction needs to be retried (for example, because a NAK was received or the target function did not respond) then the transaction is not retried until the transaction scheduler has checked all the other endpoints for active transactions first. This check ensures that an endpoint that is sending a lot of NAKs does not block other transactions on the bus. The USB controller lets you specify a limit (USB_TXINTERVAL or USB_RXINTERVAL registers) to the length of time in which NAKs may be received from a particular target before the endpoint is timed out.

Babble

If the bus is still active at the end of a frame, the USB controller assumes that the function it is connected to has malfunctioned, suspends all transactions, and generates a babble interrupt (RESET_OR_BABLE_B). The USB controller does not start a transaction until the bus is inactive for at least the minimum inter-packet delay. The controller also does not start a transaction unless it can be finished before the end of the frame.

Host Mode Reset

If the RESET bit in the USB_POWER register is set while the USB controller is in host mode, the USB controller generates reset signaling on the bus. The processor core should keep this bit set for 20 ms to ensure correct resetting of the target device. After the processor core has cleared the bit, the USB controller starts its frame counter and transaction scheduler.

Host Mode Suspend

If the SUSPEND_MODE bit in the USB_POWER register is set, the USB controller completes the current transaction then stops the transaction scheduler and frame counter. No further transactions are started and no SOF packets are generated.

To exit suspend mode, the processor core should set the RESUME_MODE bit and clear the SUSPEND_MODE bit in the USB_POWER register. While the RESUME_MODE bit is high, the USB controller generates resume signaling on the bus. After 20 ms, the processor core should clear the RESUME_MODE bit, at which point the frame counter and transaction scheduler are started.

While in suspend mode, the USB controller clock is stopped to reduce power. The SUSPEND_BE output also goes low, if enabled. This feature may be used to power-down the USB drivers. If remote wake-up is to be supported, power to the PHY must be maintained, so the USB controller can detect resume signaling on the bus.

Functional Description

The following sections describe the function of the USB OTG interface.

On-Chip Bus Interfaces

The USB controller uses two independent bus interfaces (peripheral slave and DCB/DEB master) to communicate with a processor-based sub-

system. The slave interface allows the processor core to access the control and status registers (including DMA master registers) and the endpoint FIFOs. The master interface is used to drive data into or out of the endpoint FIFOs with minimal processor core interaction.

The peripheral bus slave interface has the following characteristics.

- 16-bit wide transfers
- Wait states are asserted when FIFO accesses take place (maximum of three are possible when contention for the SRAM occurs).

The DCB/DEB bus master interface has the following characteristics:

- 16-bit wide read and write data busses
- write transfers of byte and 16-bit words are possible (byte accesses are used only for remaining bytes in a transfer)
- read transfers of 16 bits (first few or last few bytes may be discarded based on starting address and DMA count respectively)

Interface Pins

The USB OTG external interface has the pins shown in Table 26-2.

Table 26-2. USB 2.0 HS OTG Pins

Signal Name	Input/ Output	Description
USB_DP	I/O	USB D+ pin
USB_DM	I/O	USB D– pin
USB_XI	C	Clock XTAL input 1
USB_XO	C	Clock XTAL input 2
USB_ID	I	USB ID pin
USB_VBUS	I/O	USB VBUS pin
USB_V_{REF}	O	USB voltage reference source (Test purposes only)
USB_RSET	O	USB resistance set (Test purposes only)

Power and Clocking

The USB controller uses the system clock `CLK` (greater than 30 MHz required) to generate an internal clock used to clock the USB registers. The transceiver clock is a 60MHz clock sourced from the UTMI PHY and is used by the PHY interface logic and USB engine. The A 32 KHz clock (refer to the `USB_SRP_CLKDIV` register) is used for D+ pulse detection for SRP signaling by an OTG 'B' device only.

During SUSPEND and when no session is active, the clock to much of the USB controller is stopped to reduce power consumption. The clock becomes operational again when RESUME signaling is detected on the USB lines.

UTMI Interface

The interface to the on-chip PHY uses the industry-standard UTMI+ (universal transceiver macro interface) level 2. This provides full high-speed device and OTG functionality, but does not support communication to a hub.

The PHY is a mixed-signal block and includes the following:

- full-speed and high-speed drivers and receivers (single-ended and differential)
- data line pull-up and pull-down resistors
- full-speed and high-speed CDR
- VBUS and `USB_ID` level detection
- host disconnect detection
- full-speed/high-speed shift registers, NRZI encode/decode and bit-stuff encode/decode

Although the UTMI specification indicates that VBUS charging, driving and discharging be done inside the PHY, for process-restricting and power reasons, these functions are typically implemented off-chip in a separate USB charge-pump chip.

Programming Model

The following sections describe the USB OTG programming model.

Peripheral Mode Flow Charts

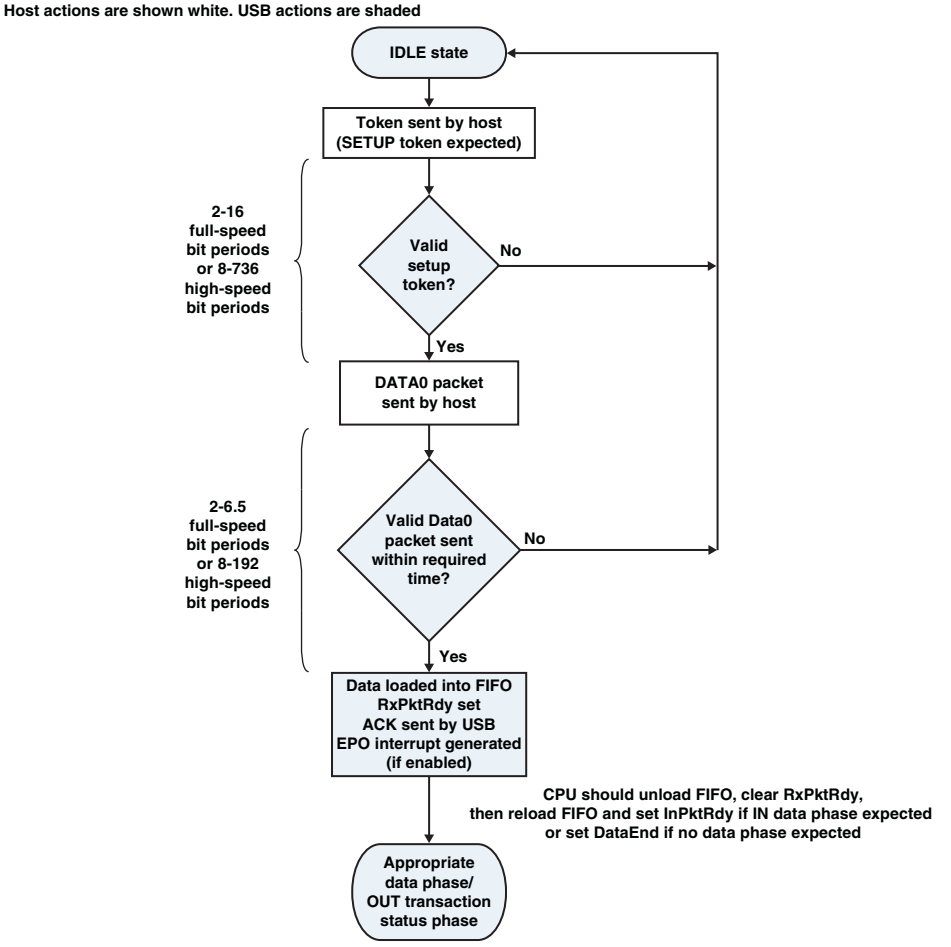


Figure 26-8. USB Control Setup Phase

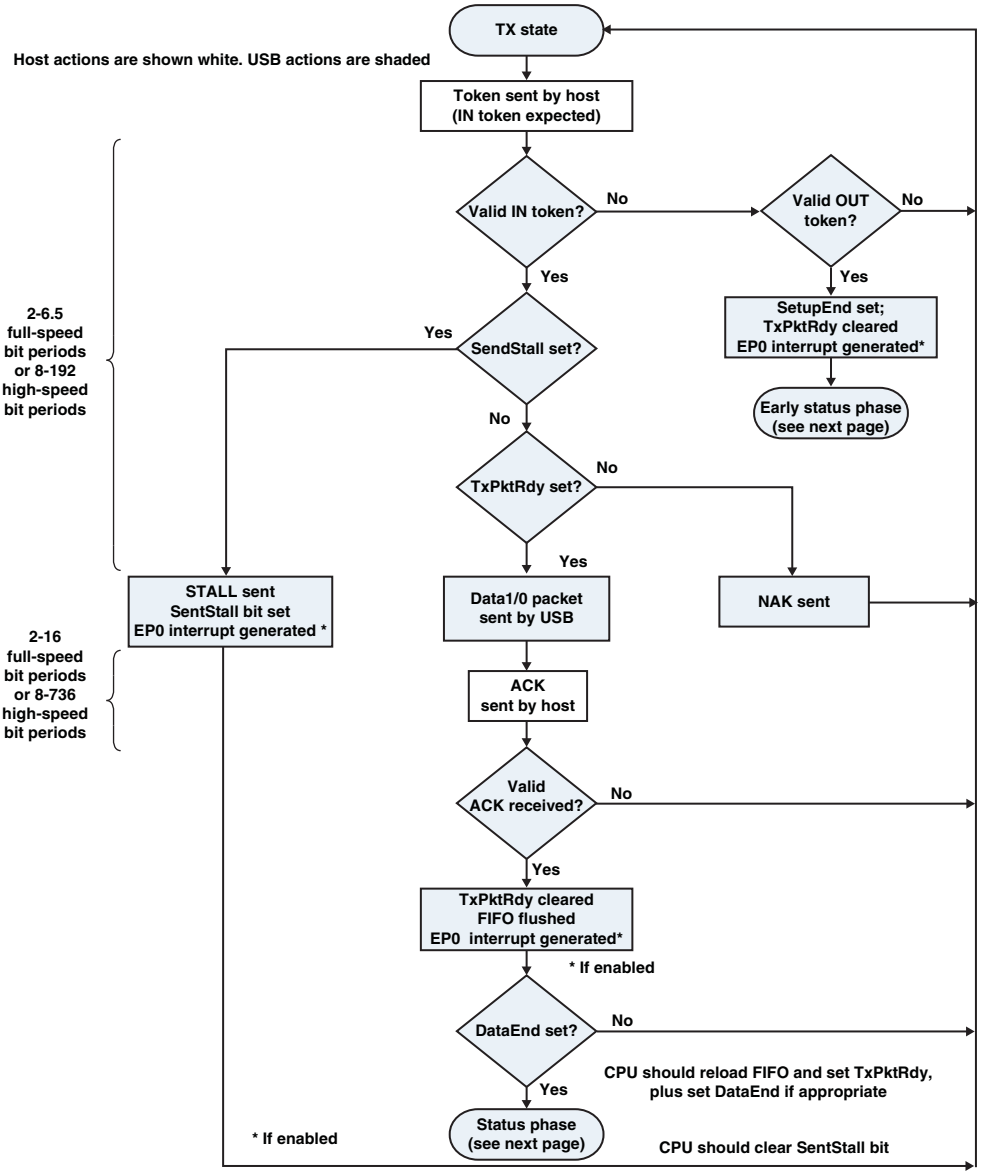


Figure 26-9. Control In Data Phase

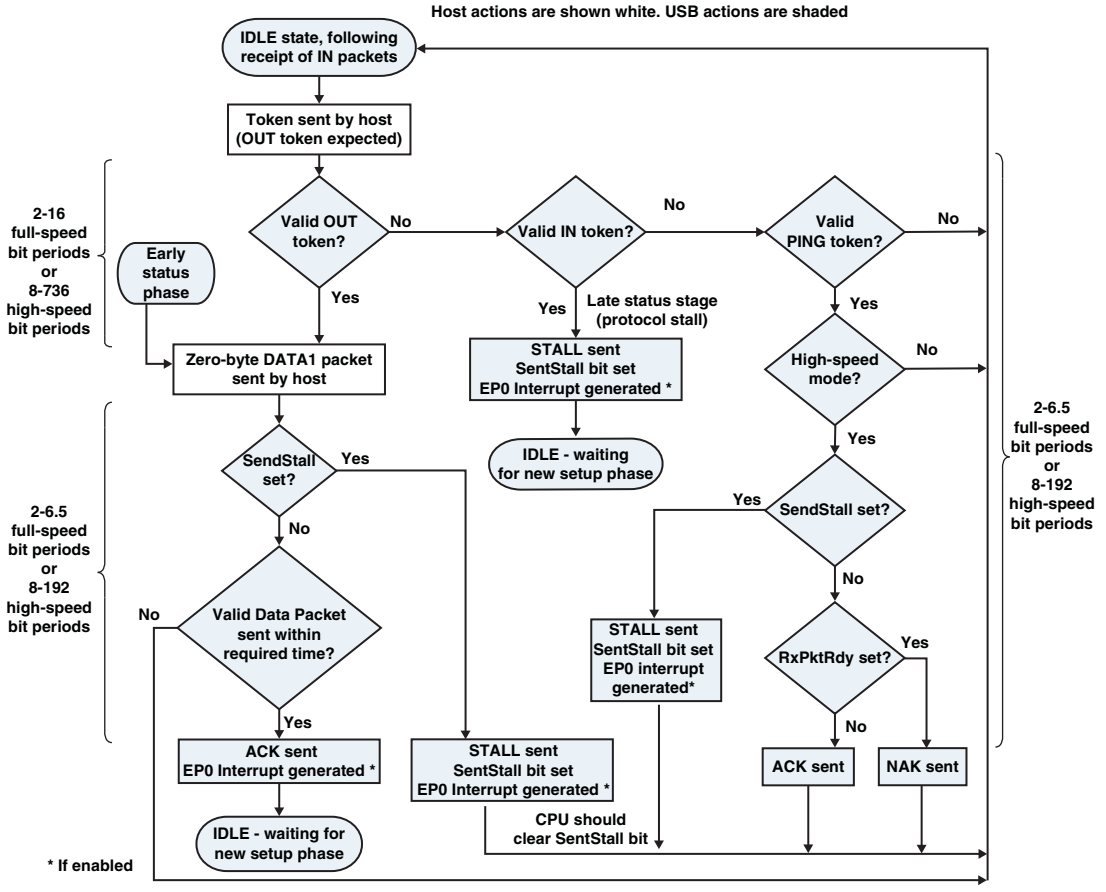


Figure 26-10. Control In Data Status Phase

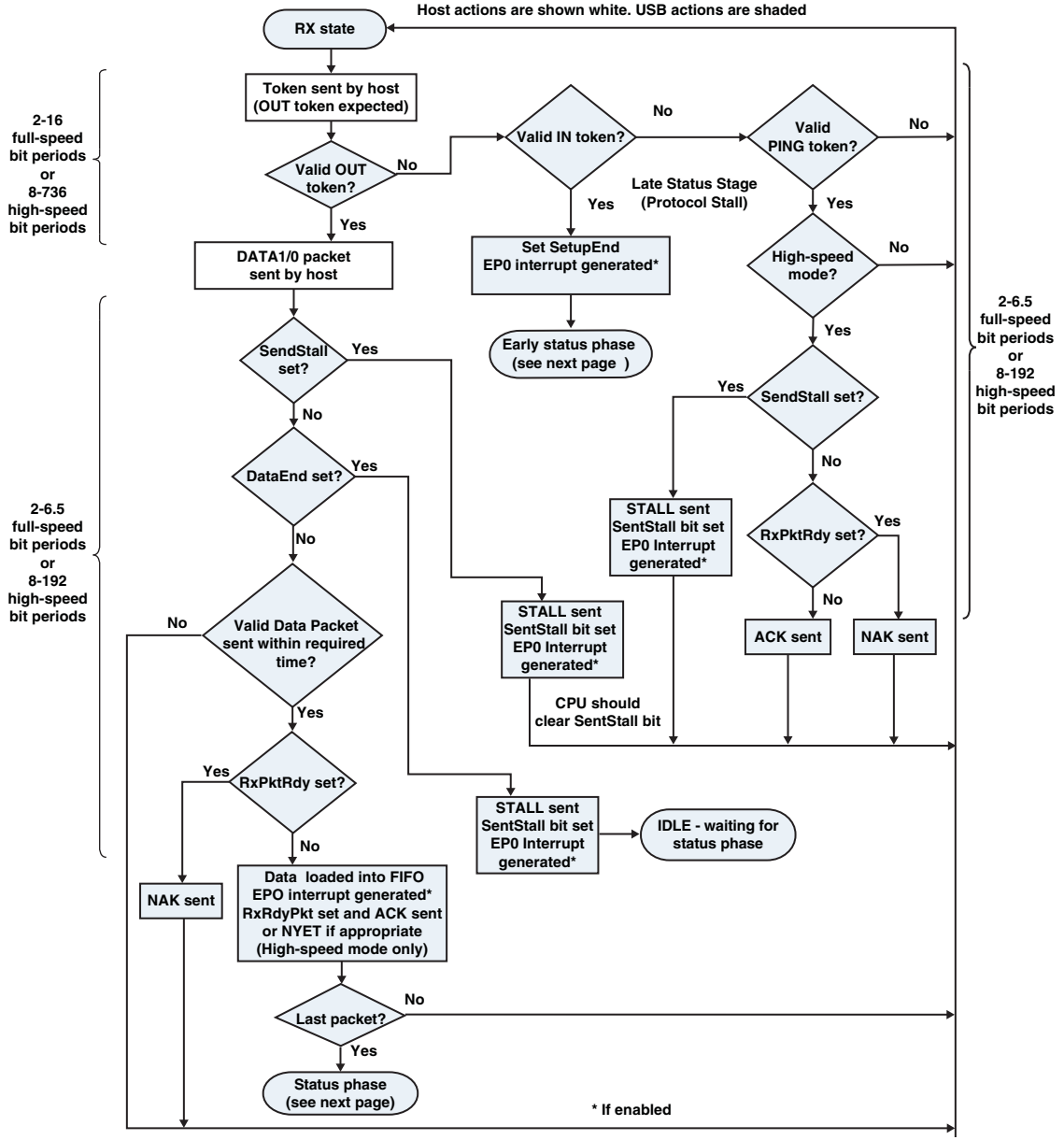


Figure 26-11. Control Out Data Phase

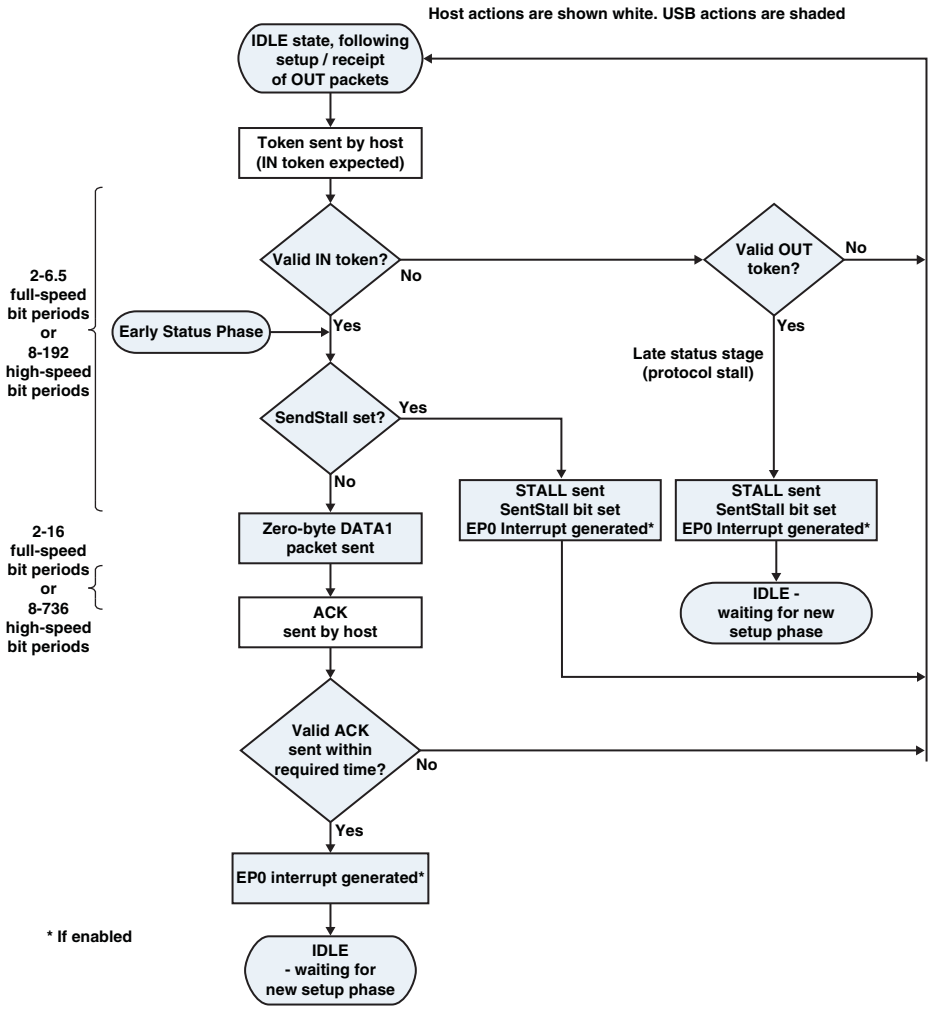

Host actions are shown white. USB actions are shaded
IDLE state, following setup / receipt of OUT packets
Token sent by host (IN token expected)
Valid IN token?
No
Valid OUT token?
No
Yes
2-6.5 full-speed bit periods or 8-192 high-speed bit periods
Early Status Phase
Yes
Late status stage (protocol stall)
SendStall set?
Yes
No
STALL sent SentStall bit set EP0 Interrupt generated*
STALL sent SentStall bit set EP0 Interrupt generated*
2-16 full-speed bit periods or 8-736 high-speed bit periods
Zero-byte DATA1 packet sent
IDLE - waiting for new setup phase
ACK sent by host
Valid ACK sent within required time?
No
Yes
EP0 interrupt generated*
* If enabled
IDLE - waiting for new setup phase

Figure 26-12. Control Out Data Status Phase

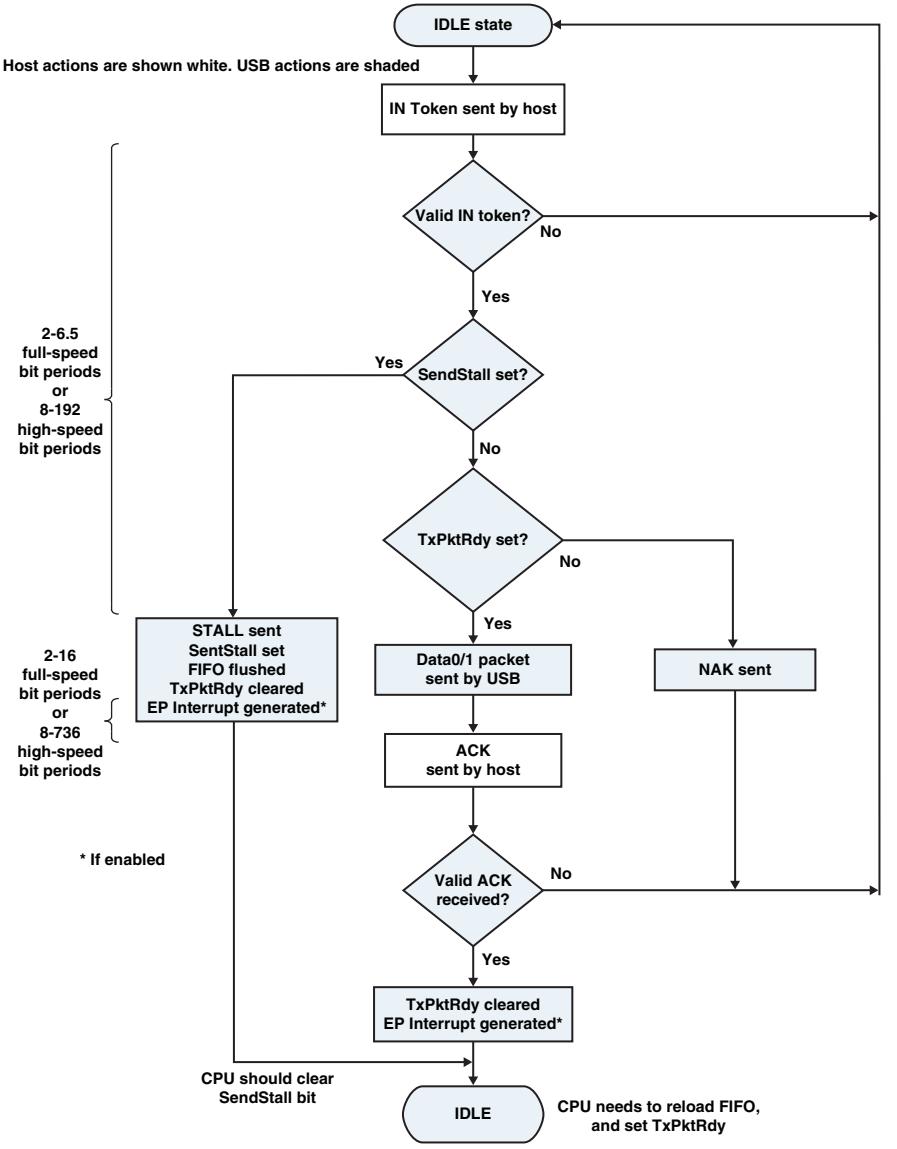


Figure 26-13. Bulk/Low Bandwidth Interrupt In Transaction

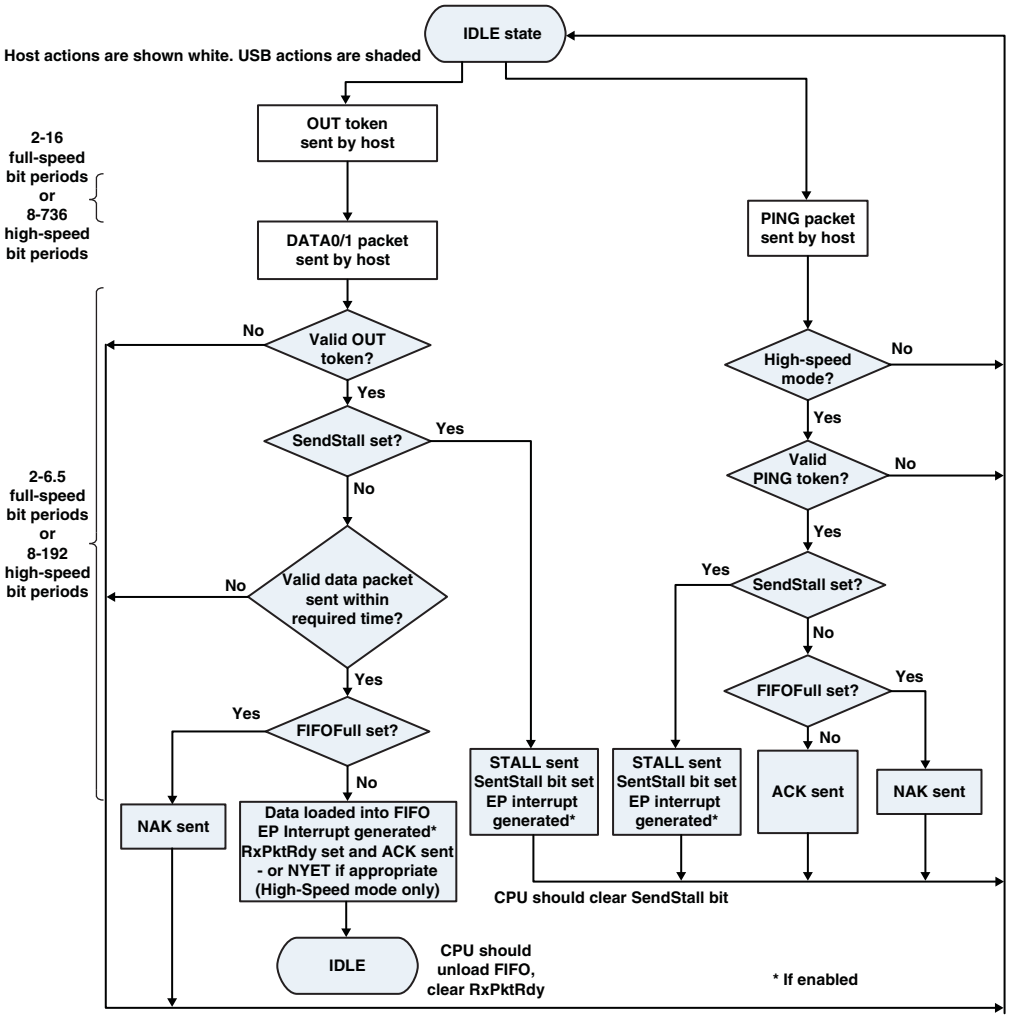


Figure 26-14. Bulk/Low Bandwidth Interrupt Out Transaction

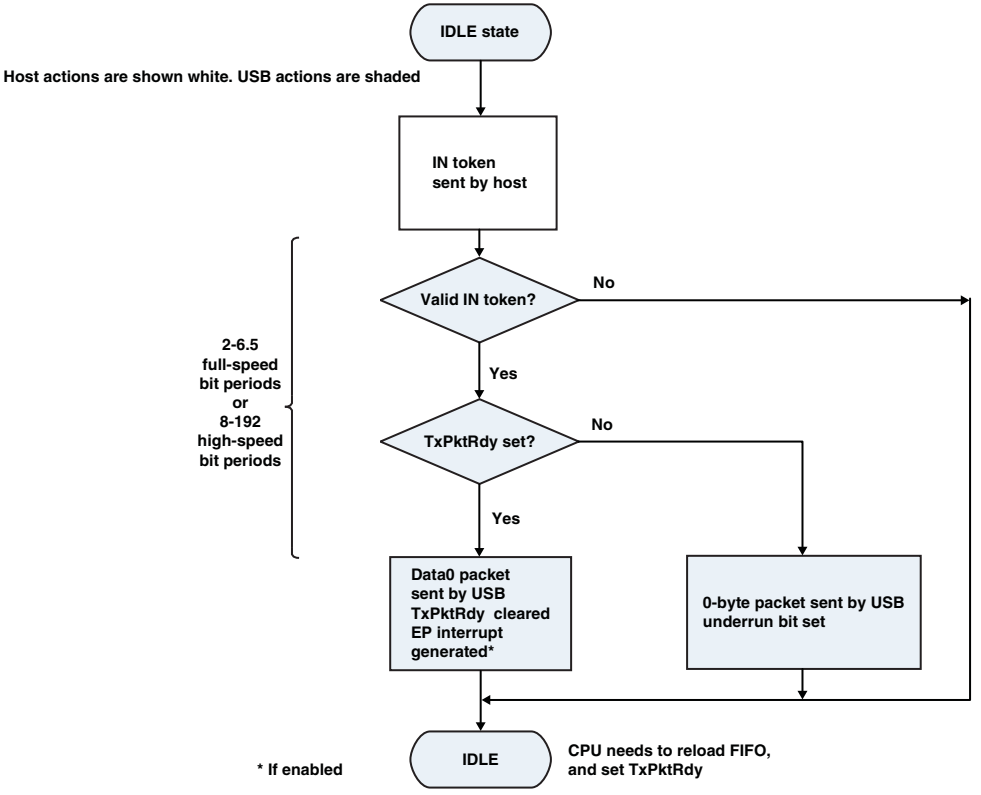


Figure 26-15. Full-speed/Low Bandwidth Isochronous In Transaction

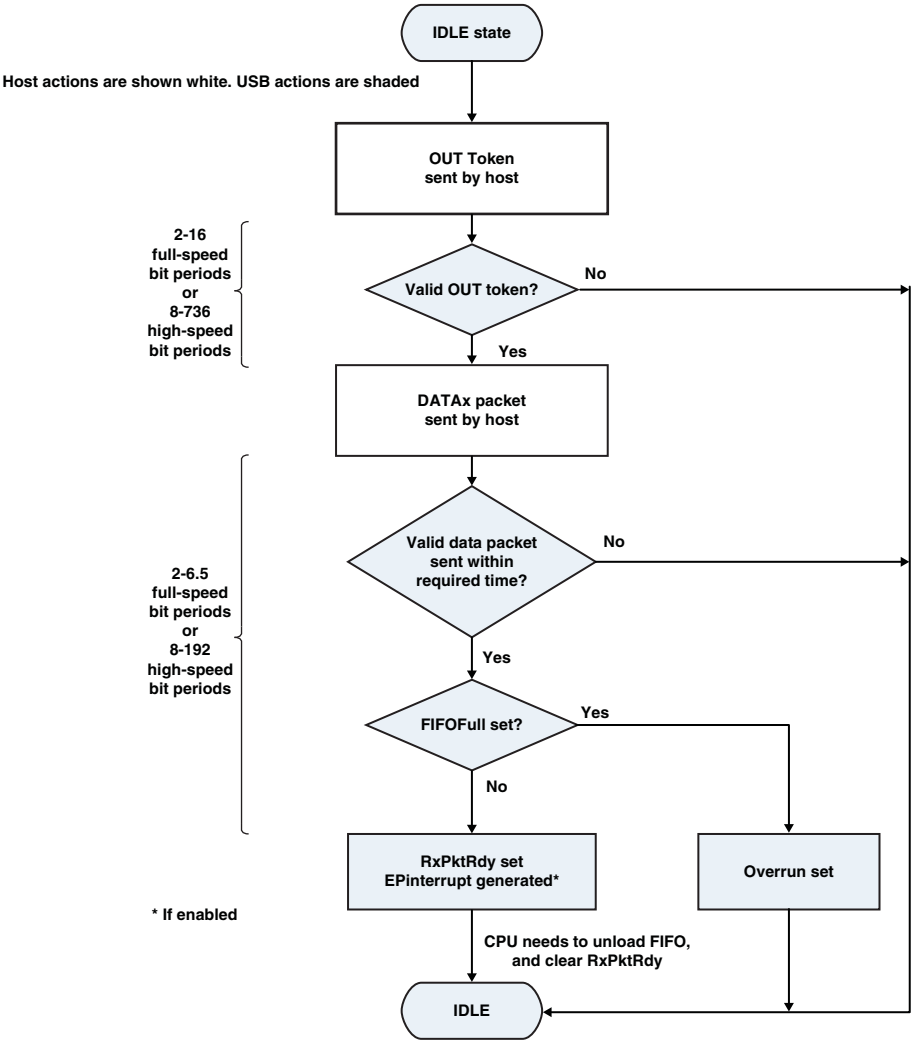


Figure 26-16. Full-speed/Low Bandwidth Isochronous Out Transaction

Host Mode Flow Charts

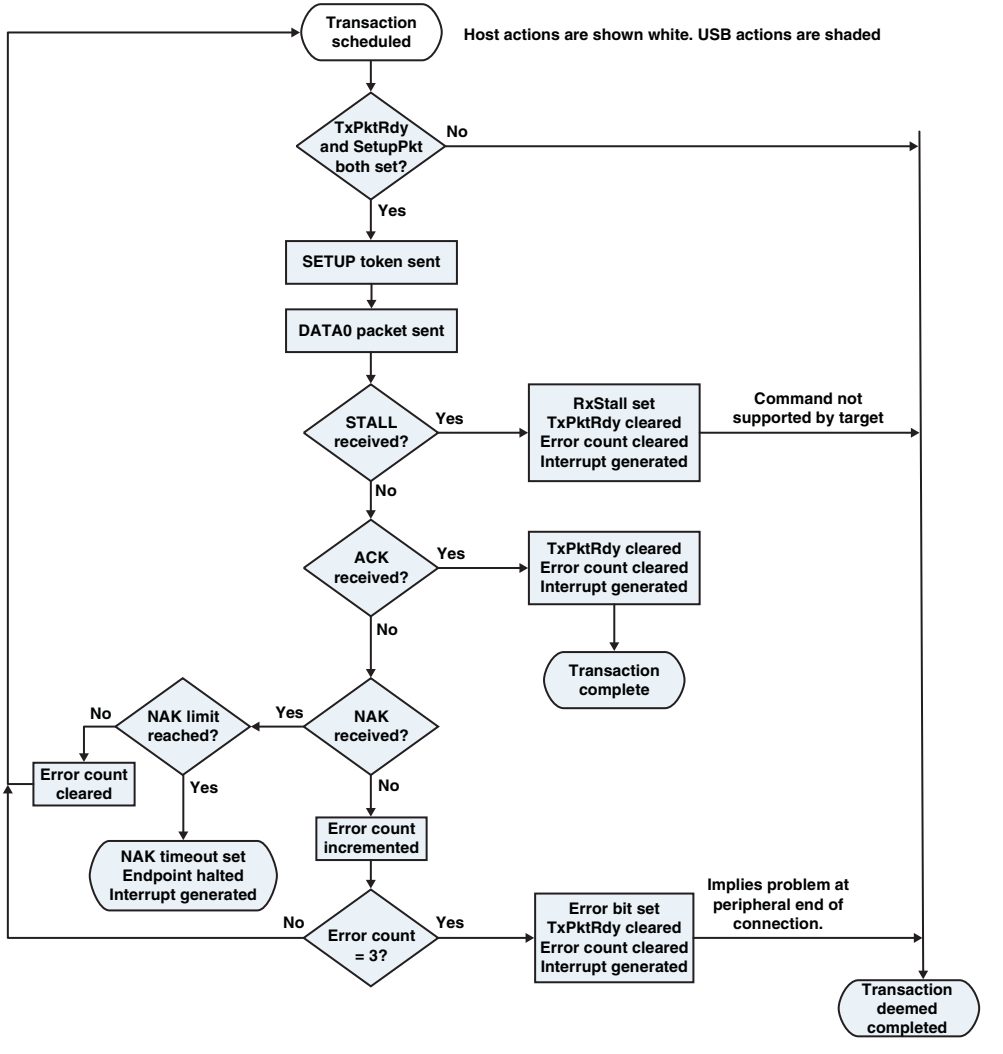


Figure 26-17. USB Control Setup Phase

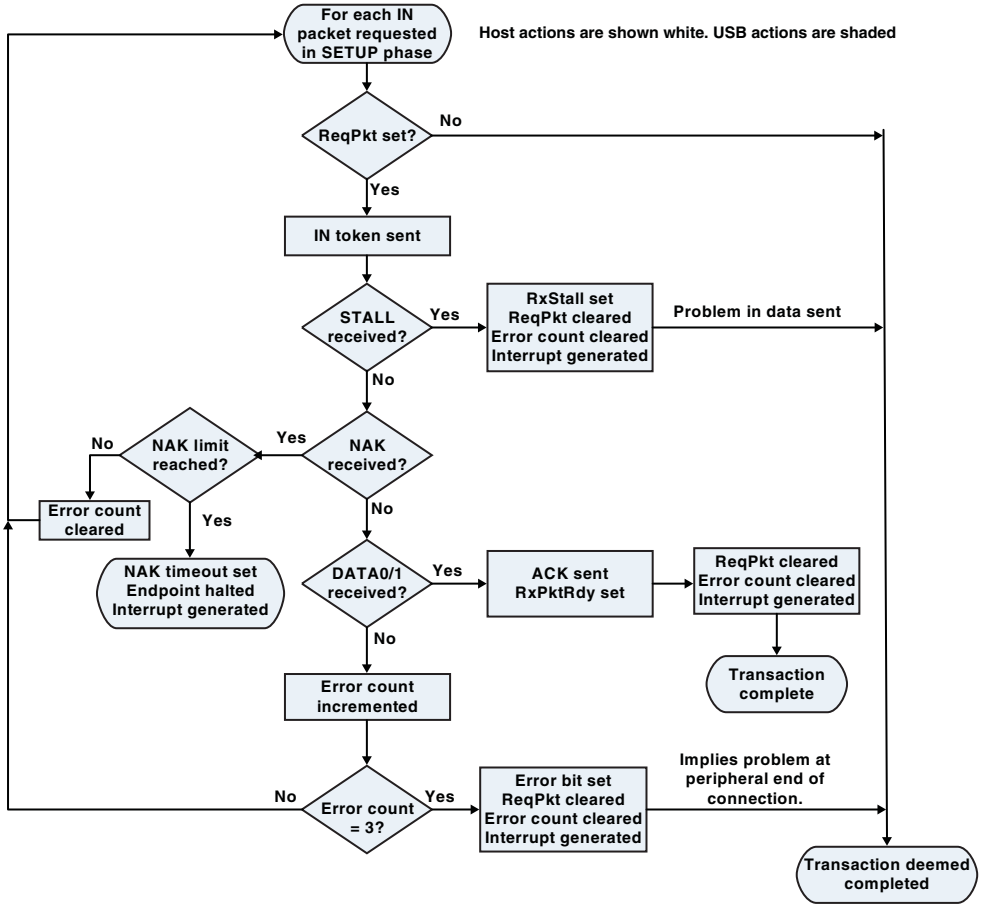


Figure 26-18. Control In Data Phase

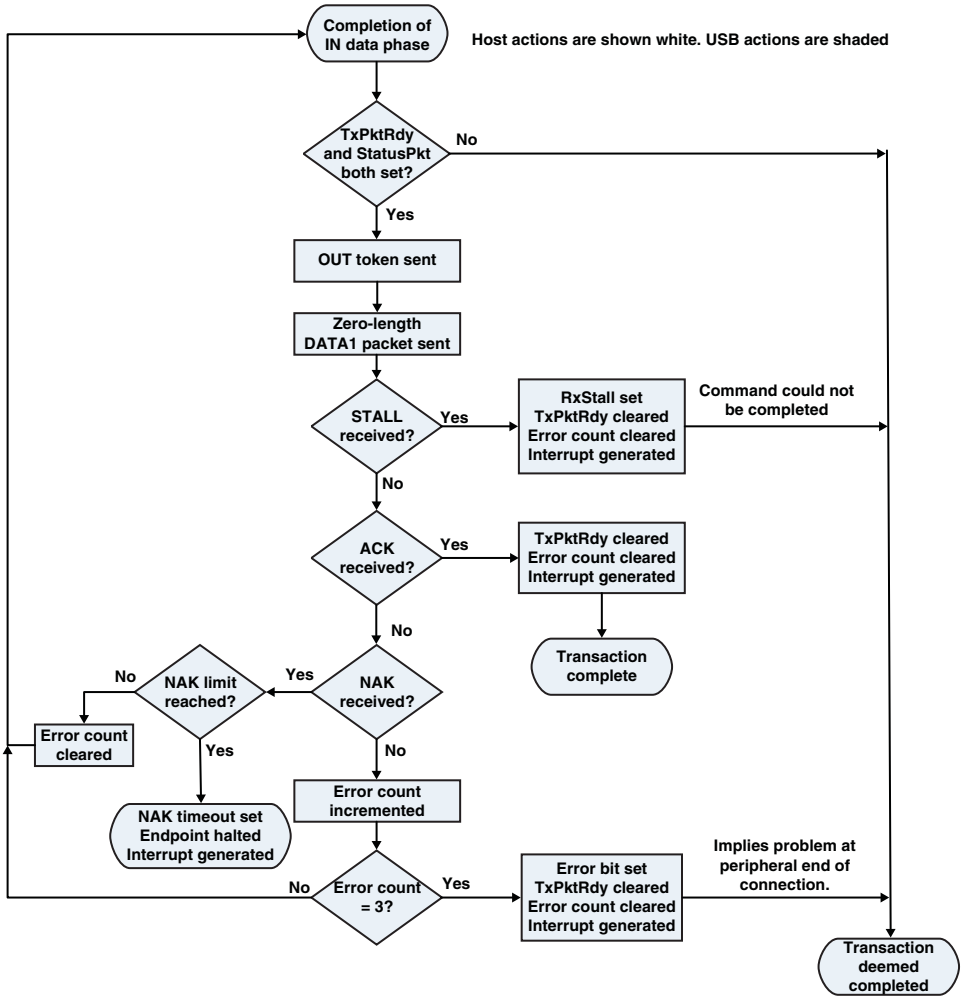


Figure 26-19. Control In Data Status Phase

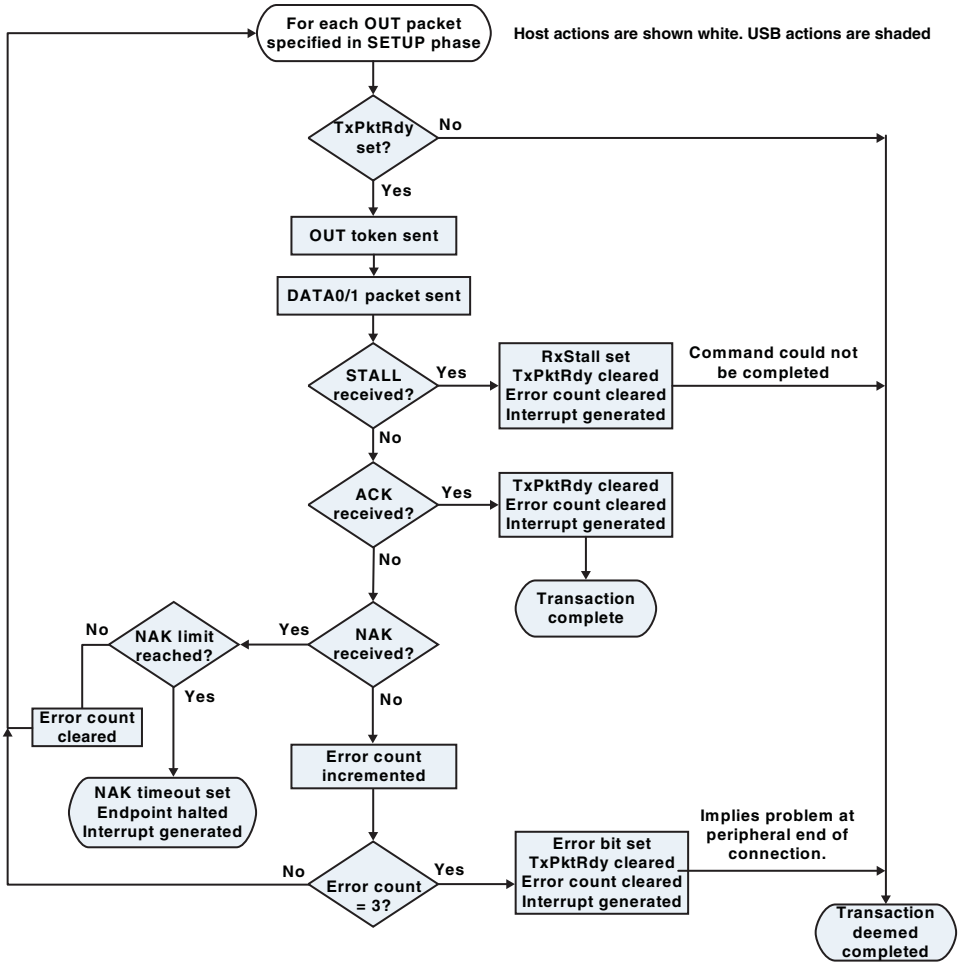


Figure 26-20. Control Out Data Phase

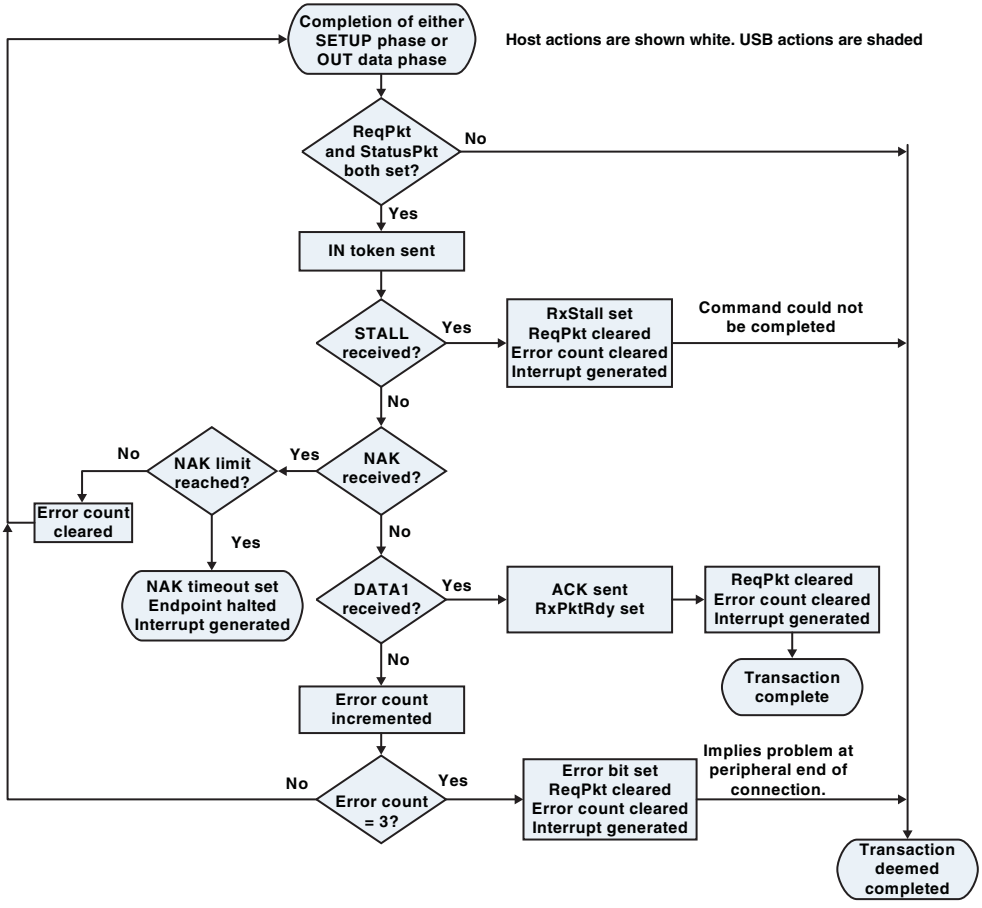


Figure 26-21. Control Out Data Status Phase

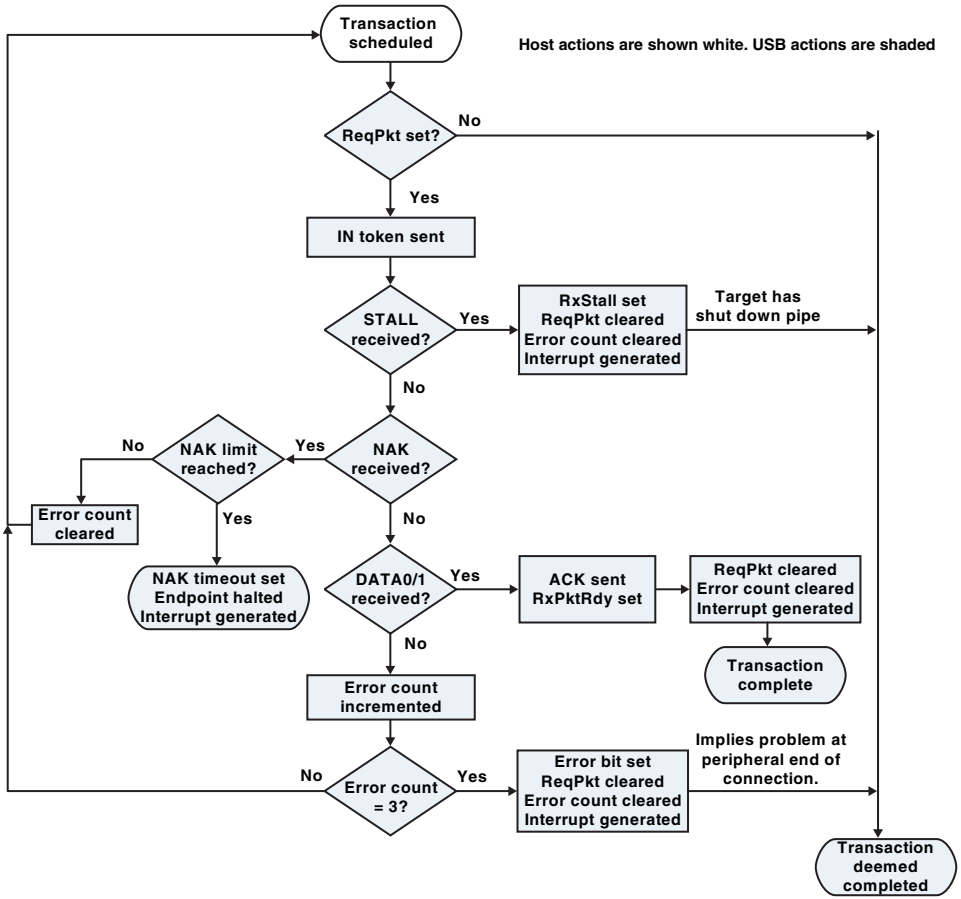


Figure 26-22. Bulk/Low Bandwidth Interrupt In Transaction

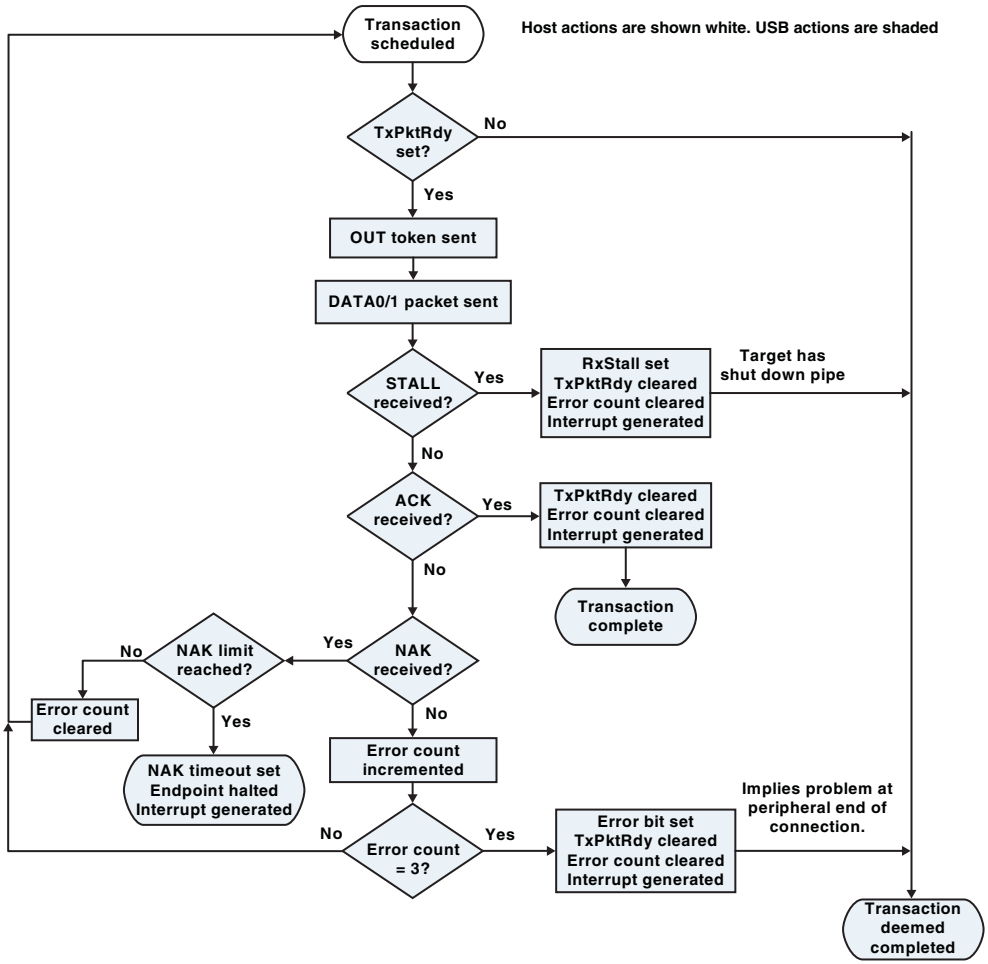


Figure 26-23. Bulk/Low Bandwidth Interrupt Out Transaction

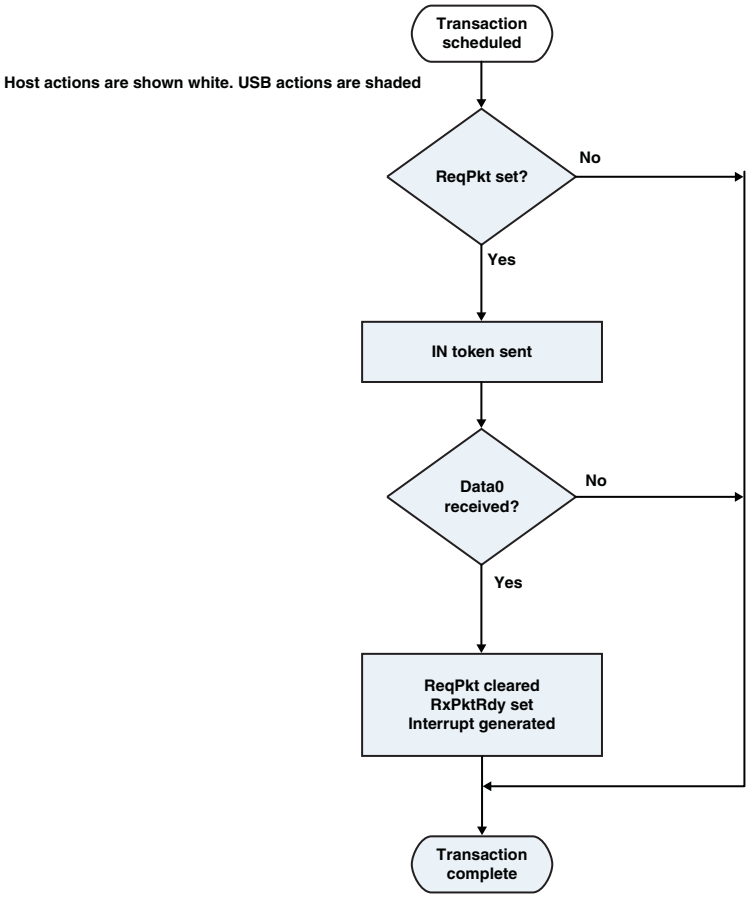


Figure 26-24. Full-speed/Low Bandwidth Isochronous In Transaction

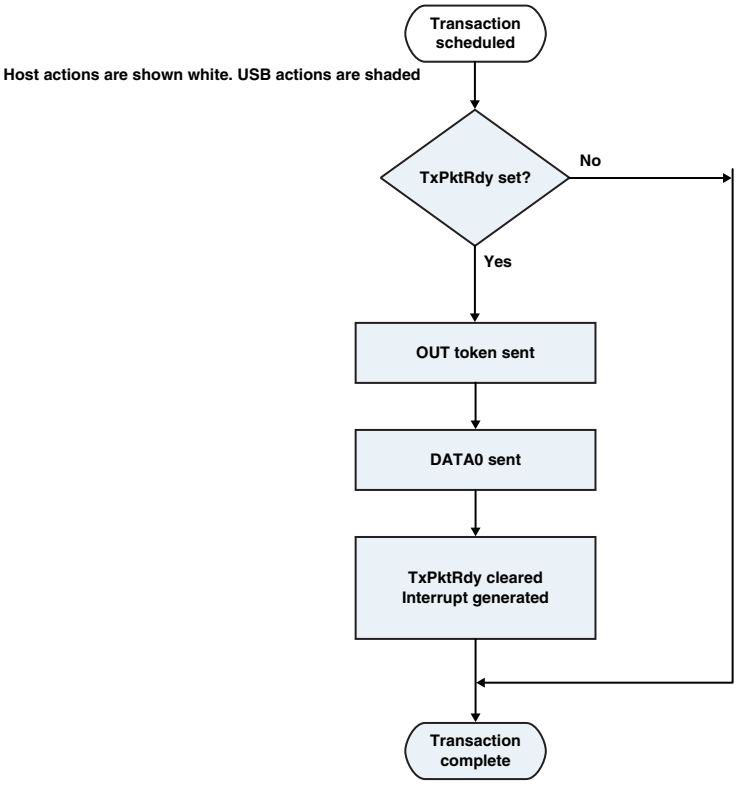


Figure 26-25. Full-speed/Low Bandwidth Isochronous Out Transaction

DMA Mode Flow Charts

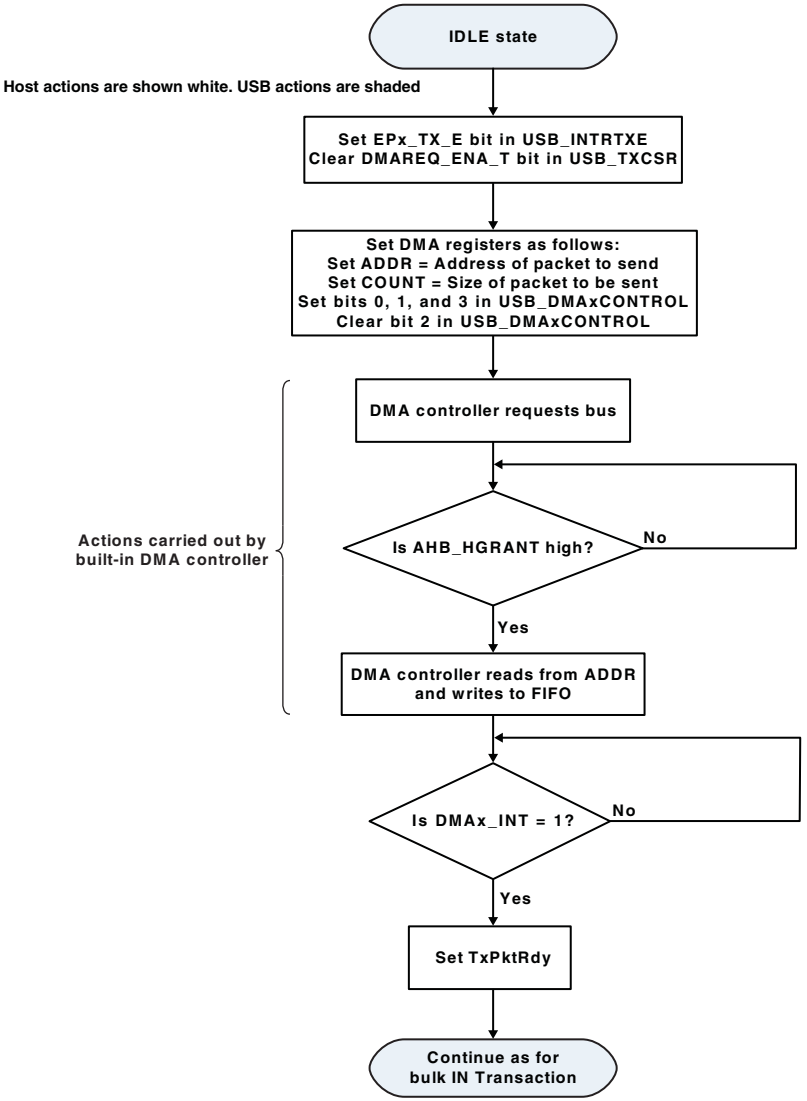

IDLE state
Host actions are shown white. USB actions are shaded
Set EPx_TX_E bit in USB_INTRTXE
Clear DMAREQ_ENA_T bit in USB_TXCSR
Set DMA registers as follows:
Set ADDR = Address of packet to send
Set COUNT = Size of packet to be sent
Set bits 0, 1, and 3 in USB_DMAxCONTROL
Clear bit 2 in USB_DMAxCONTROL
DMA controller requests bus
Actions carried out by built-in DMA controller
Is AHB_HGRANT high?
No
Yes
DMA controller reads from ADDR and writes to FIFO
Is DMAx_INT = 1?
No
Yes
Set TxPktRdy
Continue as for bulk IN Transaction

Figure 26-26. Single Packet Transmit During DMA Operation

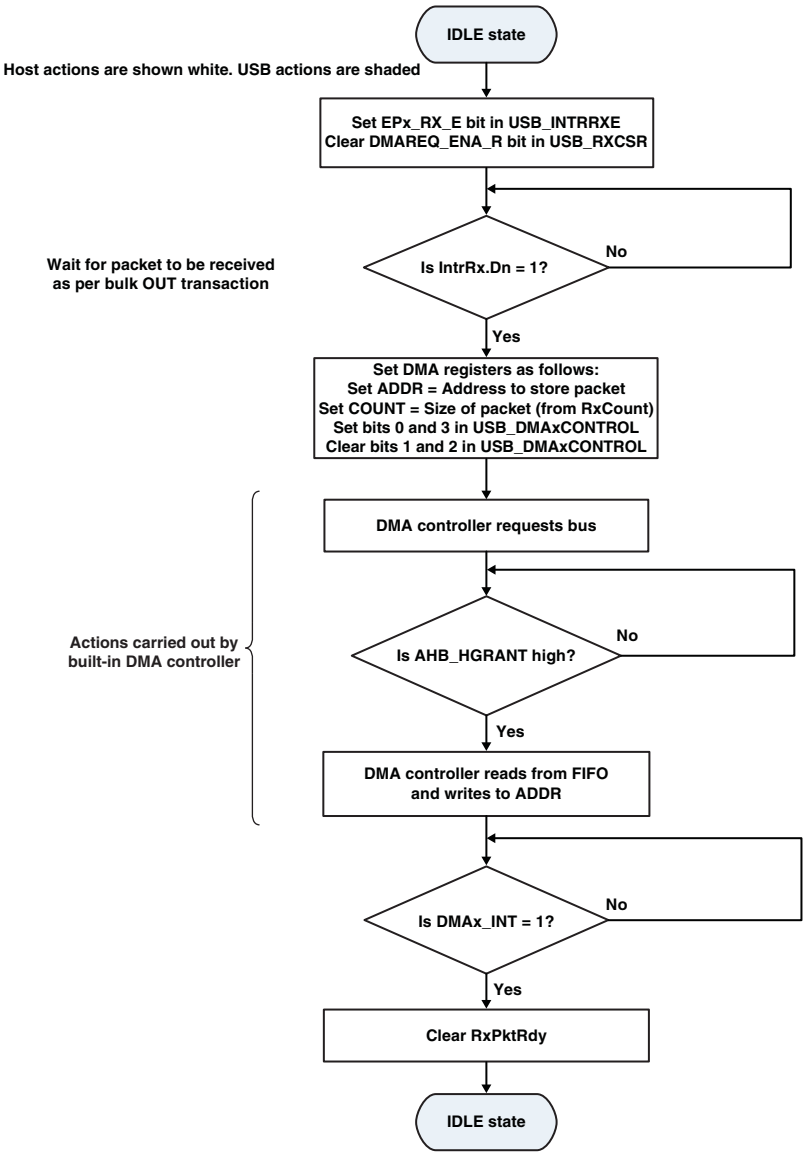

IDLE state
Host actions are shown white. USB actions are shaded
Set EPx_RX_E bit in USB_INTRRXE
Clear DMAREQ_ENA_R bit in USB_RXCSR
Wait for packet to be received
as per bulk OUT transaction
Is IntrRx.Dn = 1?
No
Yes
Set DMA registers as follows:
Set ADDR = Address to store packet
Set COUNT = Size of packet (from RxCount)
Set bits 0 and 3 in USB_DMAxCONTROL
Clear bits 1 and 2 in USB_DMAxCONTROL
DMA controller requests bus
Actions carried out by
built-in DMA controller
Is AHB_HGRANT high?
No
Yes
DMA controller reads from FIFO
and writes to ADDR
Is DMAx_INT = 1?
No
Yes
Clear RxPktRdy
IDLE state

Figure 26-27. Single Packet Receive During DMA Operation

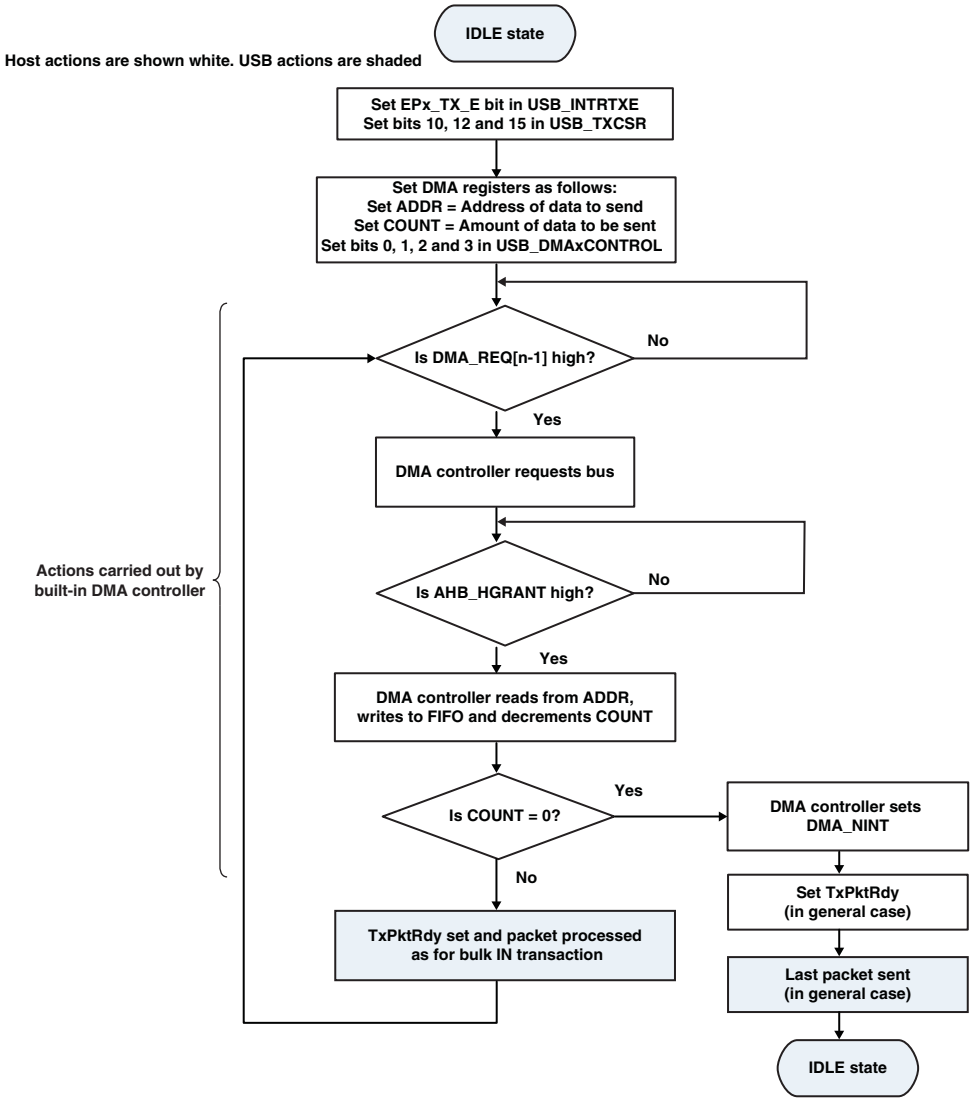


Figure 26-28. Multiple Packet Transmit During DMA Operation

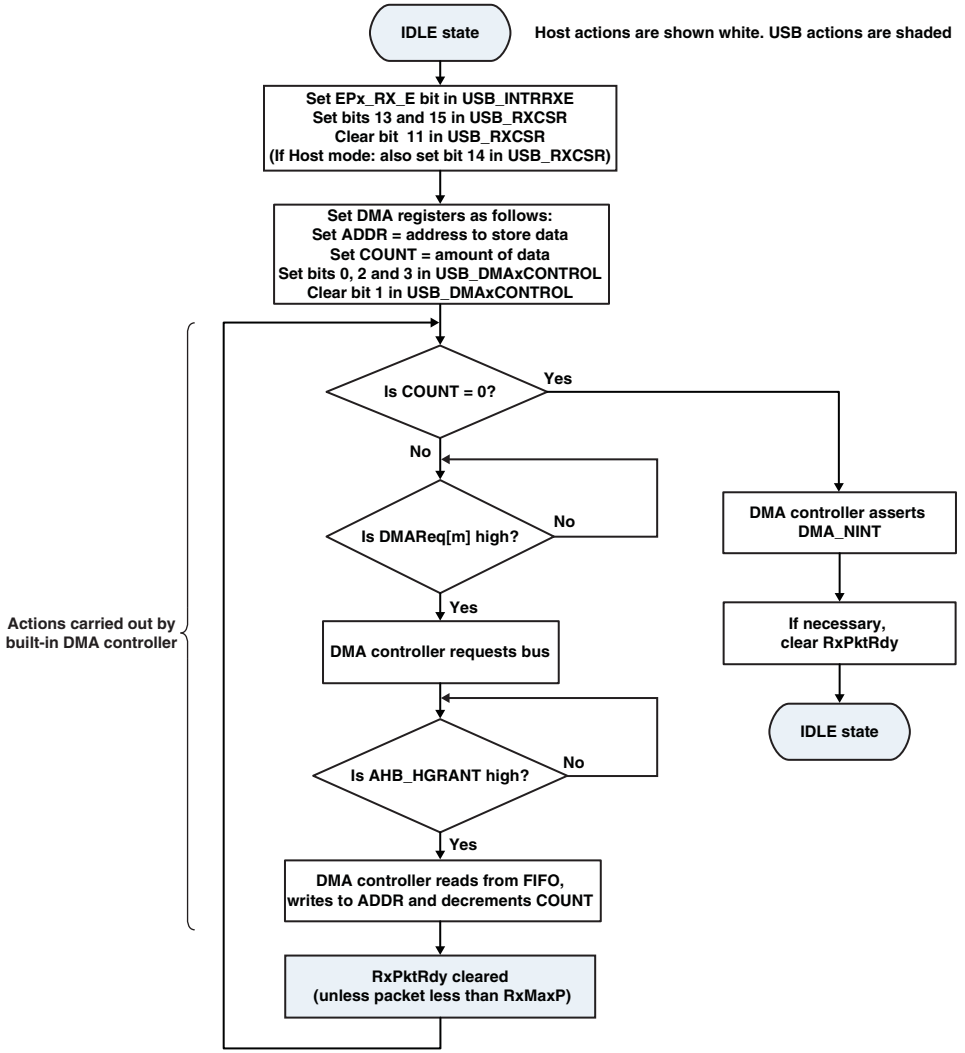


Figure 26-29. Multiple Packet Receive During DMA Operation (Data Size Known)

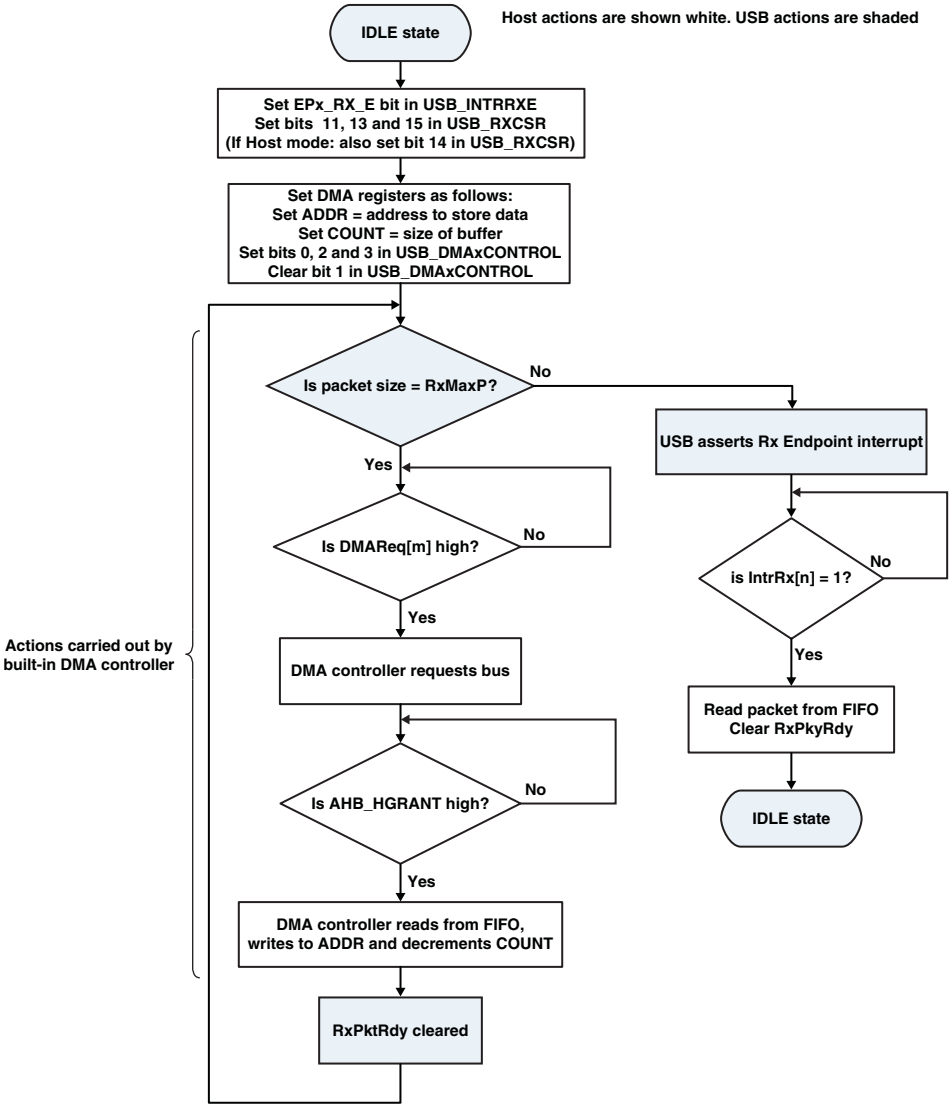


Figure 26-30. Multiple Packet Receive During DMA Operation (Data Size Not-known)

OTG Session Request

In order to conserve power, the USB on-the-go supplement allows VBUS to only be powered up when required and to be turned off when the bus is not in use.

VBUS is always supplied by the 'A' device on the bus. The USB controller determines whether it is the 'A' device or the 'B' device by sampling the USB_ID input from the PHY. This signal is pulled low when an A-type plug is sensed (signifying that the USB controller is the 'A' device), but the input is taken high when a B-type plug is sensed (signifying that the USB controller is the 'B' device).

Starting a Session

When the device containing the USB controller wants to start a session, the processor core must set the SESSION bit in the USB_OTG_DEV_CTL register. The USB controller then enables ID pin sensing. This results in the USB_ID input either being taken low if an A-type connection is detected or high if a B-type connection is detected. The B_DEVICE bit in the USB_OTG_DEV_CTL register is also set to indicate whether the USB controller has adopted the role of the 'A' device or the 'B' device.

If the USB controller is the 'A' device: The USB controller then enters host mode (the 'A' device is always the default host), and waits for VBUS to go above the VBUS valid threshold, as indicated when the VBUS1-0 bits in the USB_OTG_DEV_CTL register go to 11.

The Blackfin USB controller does not source VBUS, except when initiating SRP. As such, VBUS must be provided by an external regulator or USB charge pump.

The USB controller then waits for a peripheral to be connected. When a peripheral is detected, a connect interrupt (CONN_B bit in USB_INTRUSB) is generated (if enabled) and either the FSDEV or LSDEV bit in the USB_OTG_DEV_CTL register is set, depending on whether a full-speed periph-

eral or a low-speed peripheral was detected. The processor core should then reset this peripheral. To end the session, the processor core should clear the SESSION bit in USB_OTG_DEV_CTL.

If the USB controller is the 'B' device: The USB controller requests a session using the session request protocol defined in the USB on-the-go supplement (for example, it first asserts the DISCHRG_VBUS_START bit in USB_OTG_VBUS_IRQ to discharge VBUS). Then, when VBUS has gone below the session end threshold (as indicated by the VBUS1-0 bits in the USB_OTG_DEV_CTL register going to 00), and the line state is SE0 for greater than 2 ms, the USB controller first pulses the data line then pulses VBUS (by taking the interrupt CHRG_VBUS_START in USB_OTG_VBUS_IRQ high).

At the end of the session, the SESSION bit is cleared – usually by the USB controller but it can also be cleared by the processor core if the application software wishes to perform a software disconnect. For more information, see the description of "USB OTG Device Control (USB_OTG_DEV_CTL) Register" on page 26-132. The USB controller switches on the pull-up resistor on D+. This signals to the 'A' device to end the session.

Detecting Activity

When the other device of the OTG set-up wants to start a session, it *either* raises VBUS above the session valid threshold (if it is the 'A' device as indicated by the VBUS1-0 bits in the USB_OTG_DEV_CTL register going to 10), *or (*if it is the 'B' device) first pulses the data line then pulses VBUS. Depending on which of these actions happens, the USB controller can determine whether it is the 'A' device or the 'B' device in the current set-up and act accordingly.

If VBUS is raised above the session valid threshold, the USB controller is the 'B' device. The USB controller sets the SESSION bit in the USB_OTG_DEV_CTL register. When reset signaling is detected on the bus, a reset interrupt (RESET_OR_BABLE_B =1) is generated (if enabled) that the processor core should interpret as the start of a session. The USB control-

ler is in peripheral mode at this point as the 'B' device is the default peripheral.

At the end of the session, the 'A' device turns off the power to VBUS. When VBUS drops below the session valid threshold (as indicated by the VBUS1-0 bits in the USB_OTG_DEV_CTL register going to 01), the USB controller detects this and clears the SESSION bit to indicate that the session has ended. A disconnect interrupt (DISCON_B bit in USB_INTRUSB) is also generated (if enabled).

If data line/VBUS pulsing is detected, the USB controller is the 'A' device. The controller generates a SESSION_REQ_B interrupt (bit 6 in USB_INTRUSB, if enabled) to indicate that the 'B' device is requesting a session. The processor core should then start a session by setting the SESSION bit.

Host Negotiation/Configuration

When the USB controller is the 'A' device (USB_ID low, B_DEVICE= 0), the controller automatically enters host mode when a session starts.

When the USB controller is the 'B' device (USB_ID high, B_DEVICE= 1), the controller automatically enters peripheral mode when a session starts. The processor core can request that the USB controller become the host by setting the HOST_REQ bit in the USB_OTG_DEV_CTL register. This bit can be set either when requesting a session start by setting the SESSION bit in USB_OTG_DEV_CTL or at any time after a session has started. When the USB controller next enters suspend mode (no activity on the bus for 3 ms), and assuming the HOST_REQ bit remains set, the controller enters host mode and begins host negotiation (as specified in the USB OTG supplement), causing the PHY to disconnect the pull-up resistor on the D+ line. This should cause the 'A' device to switch to peripheral mode and to connect its own pull-up resistor. When the USB controller detects this, it generates a connect interrupt (CONN_B bit in USB_INTRUSB) if this is enabled. The controller also sets the RESET bit in the USB_POWER register to begin resetting the 'A' device. (The USB controller begins this reset sequence automati-

cally to ensure that reset is started as required within 1 ms of the 'A' device connecting its pull-up resistor). The processor core should wait at least 20 ms, then clear the RESET bit and enumerate the 'A' device.

When the USB controller-based 'B' device has finished using the bus, the processor core should put it into suspend mode by setting the SUSPEND_MODE bit in the USB_POWER register. The 'A' device should detect this and either terminate the session or revert to host mode. If the 'A' device is USB controller-based, it generates a disconnect interrupt (DISCON_B bit in USB_INTRUSB) if this is enabled.

Software Clock Control

Power consumption is minimized in the USB controller by software-controlled clock propagation. The USB_GLOBAL_CTL register is used to enable clocks to only those parts of the controller that are necessary to perform a given USB function. The GLOBAL_ENA bit must be set in order to do any operations with the USB, even including writing to other registers. Endpoint 0 control and FIFO access depends on the GLOBAL_ENA bit.

The remaining endpoint 1 – 7 TX and RX register access, transfer operation and FIFO access is dependent on the corresponding bit of USB_GLOBAL_CTL being set. State is retained in the registers when the particular endpoint clock is stopped.

Wakeup from Hibernate State

To conserve power when the chip is idle, systems often uses powerdown modes to shut down power and clocks to various parts of the chip. Hibernate state saves the most power (core clock, peripherals clocks, and internal power are off; only external power is on).

During the course of normal operation, the software can decide that the chip has been idle for a long enough period that there is no immediate need for the clocks to be active and the chip can be put into a power-down

mode such as hibernate. This period of inactivity occurs when there is a USB suspend state (idle on the bus for greater than 3 ms) or if no OTG session is valid. The SUSPEND_MODE bit (in USB_POWER) and VBUS1-0 status bits (in USB_OTG_DEV_CTL) are used to indicate these states.

Before the system software (driver) pushes processor into the hibernate state, the software has to make sure that the CSR_HBR bit (in USB_APHY_CNTRL2) is set. Setting this bit activates the non-idle activity detection logic in the PHY. Any non-idle activity on the USB bus is detected by the non-idle activity detection logic in the analog PHY. This logic wakes up the processor and generates a low to high transition on EXT_WAKE pin.

To be able to use non-idle activity detection logic as a wakeup source for the processor, enable the USB wakeup source by programming the appropriate bits in the voltage regulator control register (VR_CTL). After the processor wakes up, USB is listed as the wakeup source in the PLL status (PLL_STAT) register. The EXT_WAKE pin can be used by the external power-up sequence chip to power up SDRAM or an other external peripheral. The processor typically goes through these steps (see Figure 26-31 on page 26-83) when it comes out of hibernate state.

After the chip comes out of hibernate state, the software has to make sure that the CSR_RSTD bit of the USB_APHY_CNTRL2 register is set. This setting

deactivates the non-idle activity detection logic and ensures proper USB functionality.

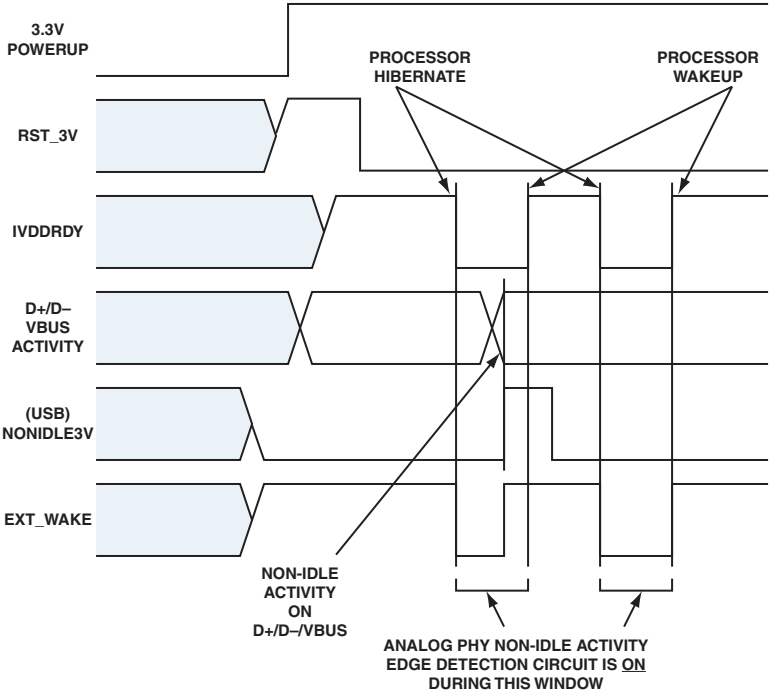


Figure 26-31. Timing Diagram of EXT_WAKE Pin

The interrupt will be asserted when either of the following events occur:

- Non-idle signaling occurs during the USB suspend state (including USB reset signaling)
- VBUS falls below the session valid threshold

Wakeup Without Re-Enumeration

When USB goes into suspend mode after 3 ms of inactivity on the D+ and D–, it is possible that the processor is pushed into the hibernate state.

Hibernate state implies that all internal power is shut down, and only the external 3.3 V power is present. And, all the clocks in the processor are shut down. If the USB were to wake up in response to non-idle activity on the D+ and D–, the USB controller would have lost the state it was in before going to hibernate. This lost state would cause the host to re-enumerate USB controller device. To prevent re-enumeration of the USB device, system software must do the following.

- Before the system software (driver) pushes the processor into hibernate, it must make sure that the state of the USB is stored in external memory flash.
- Also, the software must make sure that the CSR_HBR bit is set in the USB_APHY_CNTRL2 register.

A low to high transition on CSR_HBR generates a pulse (high) on the csr_hbr 1v signal (internal USB controller signal). This signal is used by the USB analog PHY to retain the states of the pull-up and pull-down resistors during the hibernate state. Retaining the states of the pull-up and pull-down resistors on D+ and D– implies to the host that the USB controller device is not disconnected from the USB bus.

After the system software pushes the processor into hibernate state, any non-idle activity on the USB bus is detected by the non-idle activity detection logic in the analog PHY. After the processor wakes up from the hibernate state, the processor typically goes through these steps: powering up the processor, waiting for the PLL to lock, and booting the code into L1 memory.

After code is loaded into L1 memory, it is executed. The executed code restores the state of the USB to pre-hibernate state. After the state is resumed, the analog PHY no longer needs to retain the state of the pull-ups and pull-downs on D+ and D–. The system software has to make sure that CSR_RSTD bit is set in the USB_APHY_CNTRL2 register. A low to high transition on the CSR_RSTD bit generates a pulse on the csr_rstd 1v signal (internal USB controller signal). This signal is used by the analog PHY to

prevent holding the values of pull-up and pull-down resistors. The pull-ups and pull-downs are now controlled by the USB controller. This sequence of actions (see Figure 26-32) prevents re-enumeration of the USB controller device after the processor wakes up from hibernate state.

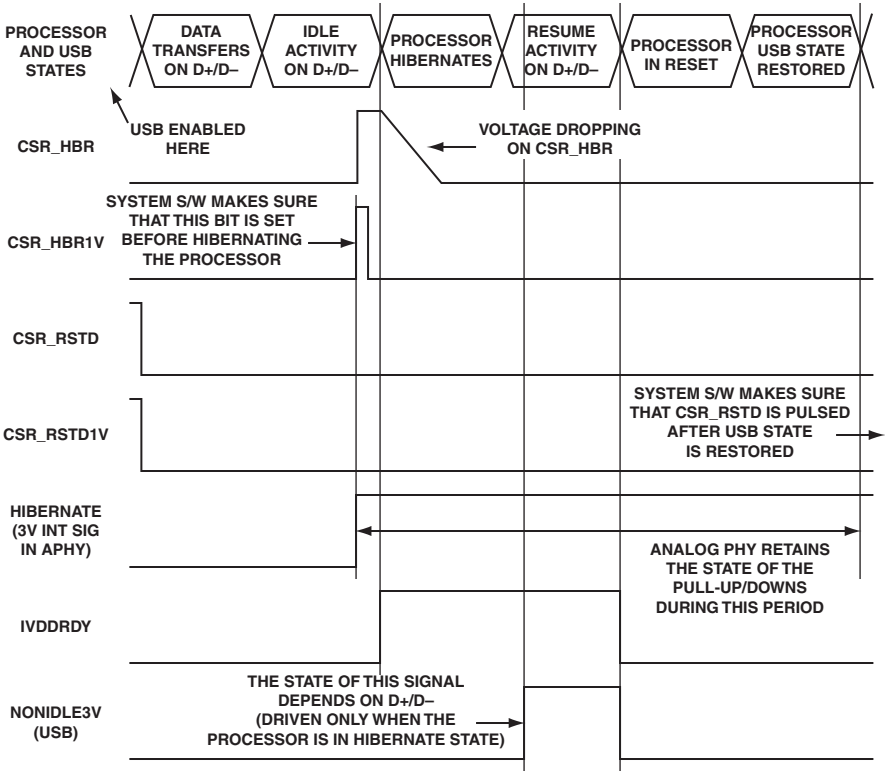

PROCESSOR AND USB STATES
DATA TRANSFERS ON D+/D–
IDLE ACTIVITY ON D+/D–
PROCESSOR HIBERNATES
RESUME ACTIVITY ON D+/D–
PROCESSOR IN RESET
PROCESSOR USB STATE RESTORED
CSR_HBR
USB ENABLED HERE
VOLTAGE DROPPING ON CSR_HBR
CSR_HBR1V
SYSTEM S/W MAKES SURE THAT THIS BIT IS SET BEFORE HIBERNATING THE PROCESSOR
CSR_RSTD
CSR_RSTD1V
SYSTEM S/W MAKES SURE THAT CSR_RSTD IS PULSED AFTER USB STATE IS RESTORED
HIBERNATE (3V INT SIG IN APHY)
ANALOG PHY RETAINS THE STATE OF THE PULL-UP/DOWNS DURING THIS PERIOD
IVDDRDY
NONIDLE3V (USB)
THE STATE OF THIS SIGNAL DEPENDS ON D+/D– (DRIVEN ONLY WHEN THE PROCESSOR IS IN HIBERNATE STATE)

Figure 26-32. Timing Diagram of the CSR_HBR/CSR_RSTD Bits

Data Transfer

Regardless of whether the USB controller is operating in host or peripheral mode, data is channeled through the endpoint FIFOs to construct packets to be sent or to be received over the USB. The RX FIFOs are used to receive OUT packets when in peripheral mode and IN packets when

operating in host mode. Similarly, the TX FIFOs are used to transmit IN packets when in peripheral mode and OUT packets as a host.

Data may be moved between the FIFOs and memory using either DMA or interrupts. Each endpoint FIFO has its own individually programmable options so that each can be set up separately. Different transfer types must be treated differently by the system. Data transfers of significant size almost certainly require DMA to move the data around; but smaller packet sizes might be handled completely by the processor.

Each data endpoint supports both double and single-buffering modes. In single-buffered operation, FIFOs are unloaded and loaded on a packet-by-packet basis. Double-buffering imposes less burden on the system by allowing two packets to be buffered in a FIFO before it is necessary to use DMA/interrupts to service the FIFO. Double-buffering mode is automatically enabled when a *MaxPktSize* is set for an endpoint that is equal to or less than half the size in bytes of that FIFO.

Loading/Unloading Packets from Endpoints

Because the peripheral bus slave interface to the USB controller provides a fixed transfer size of half words (16-bits), some additional work is required to use packet or transfer sizes that are an odd-number of bytes in length. This prevents data loss or corruption. This situation only exists for FIFO interface accesses through the processor core slave interface (DMA mastered endpoints can access individual bytes).

For TX endpoints with an odd number of bytes to be written into the FIFO, there is the possibility that an extra byte could be incorrectly written. The USB controller provides hardware counting and comparison

logic to prevent this from occurring. When writing such a packet into the USB controller, the following steps are required.

- Load the appropriate USB_TXCOUNT register with the packet/transfer size in bytes.
- Write all the data into the FIFO (using DMA or processor core) with the final half word of the transfer containing the final byte aligned to the least significant byte lane.

After a USB_TXCOUNT register is loaded with a value, it counts down the number of bytes written into that particular FIFO on each processor core or DMA write. When there is only one byte remaining in the transfer, the USB controller latches the least significant byte of the last half word.

Another use for the USB_TXCOUNT registers is to streamline DMA transfers, preventing unnecessary processor interaction in lengthy multi-packet transfers.

For RX endpoints using odd packet/transfer sizes, the software must compensate for the fact that the least significant byte lane of the final half word in the transfer is valid.

For EP0 RX transfers, if the last packet is not a multiple of four bytes it is strongly recommended that the remainder (n bytes mod 4) be unloaded from the FIFO using a special byte addressing FIFO register (EP0 FIFO address + 4). This prevents the USB controller from sending non-null data during the status phase of the control transfer.

DMA Master Channels

The USB controller provides eight DMA master channels to provide a more efficient transfer of larger amounts of data between the FIFOs and the processor core; and to free up the processor core for other tasks. Each

of these channels is configured and controlled using the DMA control registers.

Each DMA controller can operate in one of two DMA modes: 0 or 1. When operating in mode 0, the DMA controller only can be programmed to load or unload one packet, so processor intervention is required for each packet transferred over the USB. This mode can be used with any endpoint, whether it uses control, bulk, isochronous, or interrupt transactions.

When operating in DMA mode 1, the DMA controller can only be programmed to load/unload a complete bulk transfer, which can be many packets. After set up, the DMA controller loads or unloads the packets, interrupting the processor only when the transfer has completed. DMA mode 1 can only be used with endpoints that use bulk transactions. DMA mode 1 is most valuable where large blocks of data are transferred to a bulk endpoint. The USB protocol requires such packets to be split into a series of packets of *MaxPktSize* for the endpoint. Mode 1 can be used to avoid the overhead of having to interrupt the processor after each individual packet; instead the processor is only interrupted after the transfer has completed. In some cases, the block of data transferred comprises a pre-defined number of these packets that the controlling software counts through the transfer process. In other cases, the last packet in the series may be less than the maximum packet size and the receiver may use this "short" packet to signal the end of the transfer. If the total size of the transfer is an exact multiple of the maximum packet size, the transmitting software should send a null packet for the receiver to detect.

Each channel can be independently programmed for the selected operating mode.

For bulk OUT transfers using DMA mode 1, the DMA request line is asserted only when there is an edge transition of the state of the `RXPKTRDY` and a payload of `MaxPacketSize` has been received. If a data packet has been sitting in the FIFO prior to setting `DMAREQMODE1` in `USB_RXCSR`, the DMA request line will not be asserted when the DMA is enabled in the

DMAx_Control register. This will cause the data not to be read from the RX FIFO, resulting in a DMA "hang". However, since the packet arrived before DMAREQMODE and DMAREQ_ENA were enabled in USB_RXCSR, an RX interrupt will be generated for the corresponding endpoint. Therefore, the software should set the DMAREQMODE to Request Mode 0 to unload the pre-received packet. The RX interrupt service routine may look something like this:

Figure 26-33. EP RX Interrupt Service Routine

```
If USB_RXCOUNT == MaxPktSize
Switch to DMA Mode 0 and unload the packet
(in Mode 0, DMA_REQ is always asserted whenever there's data in the FIFO)
You should set the DMA_COUNT to MaxPktSize so as to unload only one packet
If AUTOCLEAR is set, you do not need to manually clear RXPKTRDY
Switch back to DMA Mode 1 and set the count to
(Total_Count – MaxPktSize)
Else
Handle as normal for case of short packet
```

DMA transfers may be 8-bit or 16-bit. All the transfer associated with one packet (with the exception of the last) must be of the same width, so that the data is consistently byte-aligned or word-aligned. The last transfer may contain fewer bytes than the previous transfers in order to complete an odd-byte or odd-word transfer.

DMA Bus Cycles

The DMA controller uses incrementing bursts of an unspecified length on the peripheral DMA bus. The controller starts a new burst when it is first granted bus mastership (whether at the start of a USB packet or when regaining the bus after being thrown off part way through a packet) and when the peripheral address starts a new 1K byte block.

When unloading packets from the FIFOs, the DMA controller requests ahead to the USB controller. Although it starts the transfer with two BUSY cycles while it is getting the first word from the FIFO, all subsequent words of the packet are immediately available. No further BUSY cycles are required. The DMA controller is associated with a two-word buffer, so no data is lost if it loses bus mastership part way through unloading a packet. When bus mastership is regained, it can continue unloading the packet without adding any BUSY cycles.

The DMA start address (written to the DMAxADDR) must be word aligned.

Split transactions and retries are supported.

Transferring Packets Using DMA

Use of the DMA master channels to access the USB controller FIFOs requires that both the appropriate channel and the endpoint be programmed appropriately. Many variations are possible. The following sections detail the standard setups used for the basic actions of transferring individual packets and multiple packets.

Individual Packet: RX Endpoint

The transfer of individual packets is normally carried out using DMA mode 0. The USB controller RX endpoint is programmed as follows:.

1. The relevant EPx_RX_E bit in the USB_INTRRXE register is set to 1.
2. The DMA_ENA bit of the appropriate USB_RXCSR register is set to 0. (There is no need to set the USB controller to support DMA for this operation.)
3. When a packet is received by the USB controller, it generates the appropriate endpoint interrupt (using USB_INTRRX). The processor should then program the appropriate DMA master channel as follows:
    - DMAxADDR: memory address to store packet
    - USB_DMAxCOUNT: size of packet (determined by reading the USB controller USB_RXCOUNT register)
    - USB_DMAxCONTROL: INT_ENA = 1, DMA_ENA = 1, DIRECTION = 0, DMAREQMODE_R = 0

The DMA controller then requests bus mastership and transfers the packet to memory. It interrupts the processor when it has completed the transfer. The processor should then clear the RXPKTRDY bit in the USB_RXCSR register.

Individual Packet: TX Endpoint

Again using DMA mode 0, a USB controller TX endpoint is programmed as follows.

1. The relevant EPx_TX_E bit in the USB_INTRTXE register is set to 1.
2. The DMA_ENA bit of the appropriate USB_TxCSR register is set to 0. (There is no need to set the USB controller to support DMA for this operation.)
3. When the FIFO can accommodate data, the USB controller interrupts the processor with the appropriate TX endpoint interrupt. The processor should then program the DMA channel as follows:
    - DMAxADDR: memory address of packet to send
    - USB_DMAxCOUNT: size of packet to be sent
    - USB_DMAxCONTROL: INT_ENA = 1, DMA_ENA = 1, DIRECTION = 1, DMAREQMODE_T = 0

The DMA controller then requests bus mastership and transfers the packet to the USB controller FIFO. When it has completed the transfer, it generates a DMA interrupt. The processor should then set the TXPKTRDY bit in the USB_TXCSR register.

Multiple Packets: RX Endpoint

Multiple packets normally are transferred using DMA mode 1. The DMA controller is programmed using the DMA registers:

- DMAxADDR: memory address of the buffer in which to store transfer
- USB_DMAxCOUNT: maximum size of data buffer
- USB_DMAxCONTROL: INT_ENA = 1, DMA_ENA = 1, DIRECTION = 0, DMAREQMODE_R = 1

The USB controller RX endpoint should now be programmed as follows:

1. The relevant EPx_RX_E bit in the USB_INTRRXE register is set to 1.
2. The AUTOCLEAR_R, DMAREQ_ENA_R and DMAREQMODE_R bits of the appropriate USB_RXCSR register is set to 1. In host mode, the AUTOREQ_RH and DMAREQMODE_RH bits should also be set to 1.

As each packet is received by the USB controller, the DMA master channel requests bus mastership and transfers the packet to memory. With AUTOCLEAR_R set, the USB controller automatically clears its RXPKTRDY bit. This process continues automatically until the USB controller receives a short packet (one of less than the maximum packet size for the endpoint) signifying the end of the transfer. This short packet is not transferred by the DMA controller: instead the USB controller interrupts the processor by generating the appropriate endpoint interrupt. The processor can then read the USB_RXCOUNT register to see the size of the short packet and either unload it manually or reprogram the DMA controller in mode 0 to unload the packet.

The DMAxADDR register is incremented as the packets are unloaded, so the processor can determine the size of the transfer by comparing the current value of DMAxADDR with the start address of the memory buffer.

If the size of the transfer exceeds the data buffer size, the DMA controller stops unloading the FIFO and interrupts the processor.

Multiple Packets: TX Endpoints

Using DMA mode 1 for a TX endpoint, the DMA controller is programmed as follows:

- DMAxADDR: memory address of data block to send
- USB_DMAxCOUNT: size of data block
- USB_DMAxCONTROL: INT_ENA = 1, DMA_ENA = 1, DIRECTION = 1, DMAREQMODE_T = 1

The USB controller TX endpoint is programmed as follows:

1. The relevant EPx_TX_E bit in the USB_INTRTXE register is set to 1.
2. The AUTOSET_T and DMA_ENA bits of the appropriate USB_EP_NIx_TXCSR register is set to 1.

When the FIFO in the USB controller becomes available, the DMA controller requests bus mastership and transfers a packet to the FIFO. With AUTOSET_T set, the USB controller automatically sets the TXPKTRDY bit. This process continues until the entire data block is transferred to the USB controller. The DMA controller then interrupts the processor by taking DMAx_INT low. If the last packet to be loaded was less than the maximum packet size for the endpoint, the TXPKTRDY bit is not set for this packet; the processor should respond to the DMA interrupt by setting the TXPKTRDY bit to allow the last short packet to be sent. If the last packet to be loaded was of the maximum packet size, then the action to take depends on whether the transfer is under the control of an application such as the mass storage software on Windows system that keeps count of the individual packets sent. If the transfer is not under such control, the processor should still respond to the DMA interrupt by setting the TXPKTRDY bit. This has the effect of sending a null packet for the receiving software to interpret as indicating the end of the transfer.

USB OTG Registers

The USB OTG has a number of memory-mapped registers (MMRs) that regulate its operation. Descriptions and bit diagrams for most of these registers are provided in the following sections. See Table A-26 on page A-27 for a complete list of USB-OTG registers and their addresses.

USB Global Control (USB_GLOBAL_CTL) Register

The `USB_GLOBAL_CTL` register (see Figure 26-34) enables software control of the internal clocking of the USB. This control permits reducing power

consumption by minimizing switching activity in endpoint logic, which is not required for use.

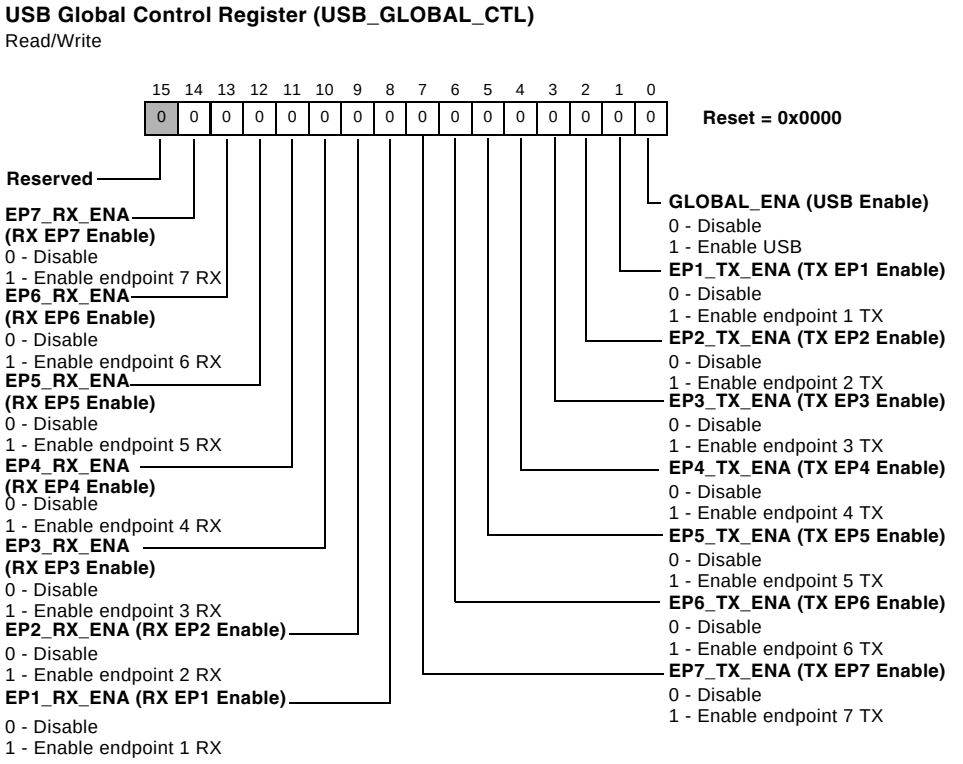

USB Global Control Register (USB_GLOBAL_CTL)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Reserved
EP7_RX_ENA
(RX EP7 Enable)
0 - Disable
1 - Enable endpoint 7 RX
EP6_RX_ENA
(RX EP6 Enable)
0 - Disable
1 - Enable endpoint 6 RX
EP5_RX_ENA
(RX EP5 Enable)
0 - Disable
1 - Enable endpoint 5 RX
EP4_RX_ENA
(RX EP4 Enable)
0 - Disable
1 - Enable endpoint 4 RX
EP3_RX_ENA
(RX EP3 Enable)
0 - Disable
1 - Enable endpoint 3 RX
EP2_RX_ENA (RX EP2 Enable)
0 - Disable
1 - Enable endpoint 2 RX
EP1_RX_ENA (RX EP1 Enable)
0 - Disable
1 - Enable endpoint 1 RX
GLOBAL_ENA (USB Enable)
0 - Disable
1 - Enable USB
EP1_TX_ENA (TX EP1 Enable)
0 - Disable
1 - Enable endpoint 1 TX
EP2_TX_ENA (TX EP2 Enable)
0 - Disable
1 - Enable endpoint 2 TX
EP3_TX_ENA (TX EP3 Enable)
0 - Disable
1 - Enable endpoint 3 TX
EP4_TX_ENA (TX EP4 Enable)
0 - Disable
1 - Enable endpoint 4 TX
EP5_TX_ENA (TX EP5 Enable)
0 - Disable
1 - Enable endpoint 5 TX
EP6_TX_ENA (TX EP6 Enable)
0 - Disable
1 - Enable endpoint 6 TX
EP7_TX_ENA (TX EP7 Enable)
0 - Disable
1 - Enable endpoint 7 TX

Figure 26-34. USB Global Control Register

Before an endpoint can be used for transfer on USB it must first be activated by setting the appropriate bit in the USB_GLOBAL_CTL register. The GLOBAL_ENA bit must be set any time the USB controller is required for use. The GLOBAL_ENA bit also brings the USB PHY and USB PLL out of reset state. The USB PLL locks with the frequency multiplier value programmed in the USB_PLLOSC_CTRL register. When USB_GLOBAL_CTL is not configured, the behavior of the USB controller is undefined and writes into CSR registers and FIFOs are not committed. It is not possible to

access an endpoint FIFO location when that endpoint is not activated in this register. Similarly, the GLOBAL_ENA bit is required for access to the endpoint 0 FIFO locations. For more information on the USB controller clocking scheme, see "Power and Clocking" on page 26-53.

Bit 15, which is marked as reserved in Figure 26-34, implements the test mode timer reduction. When set, this bit reduces the values used in the timers internal to the USB protocol block in order to drastically reduce the simulation time. This bit should only be set for simulation purposes, because setting it causes incorrect USB behavior if set during normal operation.

USB Power Management (USB_POWER) Register

The USB_POWER register (see Figure 26-35) controls suspend and resume signaling and controls some operational aspects of the USB controller.

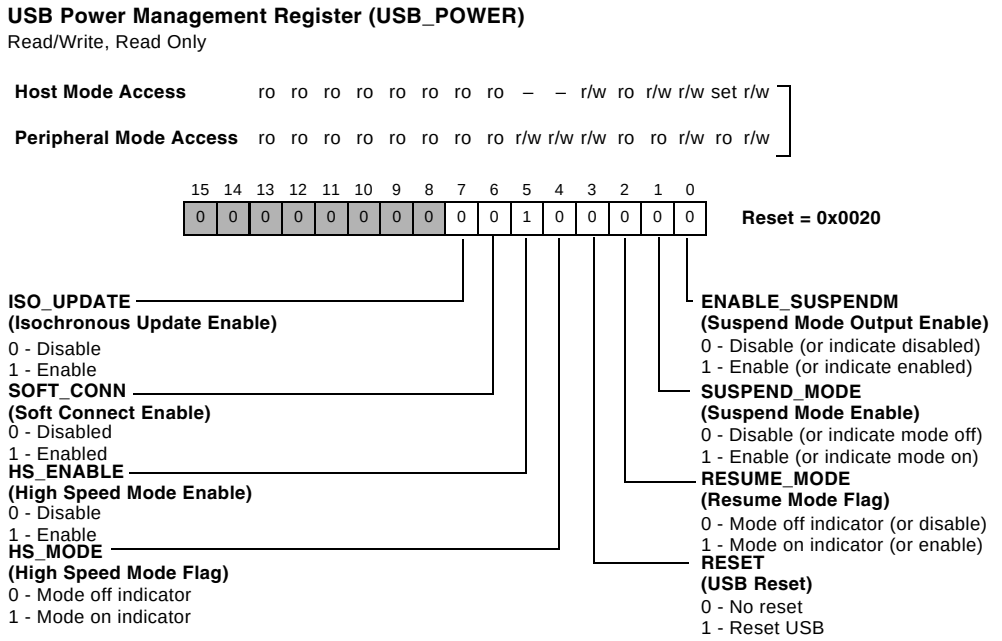

USB Power Management Register (USB_POWER)
Read/Write, Read Only
Host Mode Access ro ro ro ro ro ro ro ro – – r/w ro r/w r/w set r/w
Peripheral Mode Access ro ro ro ro ro ro ro ro r/w r/w r/w ro ro r/w ro r/w
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
Reset = 0x0020
ISO_UPDATE
(Isochronous Update Enable)
0 - Disable
1 - Enable
SOFT_CONN
(Soft Connect Enable)
0 - Disabled
1 - Enabled
HS_ENABLE
(High Speed Mode Enable)
0 - Disable
1 - Enable
HS_MODE
(High Speed Mode Flag)
0 - Mode off indicator
1 - Mode on indicator
ENABLE_SUSPENDM
(Suspend Mode Output Enable)
0 - Disable (or indicate disabled)
1 - Enable (or indicate enabled)
SUSPEND_MODE
(Suspend Mode Enable)
0 - Disable (or indicate mode off)
1 - Enable (or indicate mode on)
RESUME_MODE
(Resume Mode Flag)
0 - Mode off indicator (or disable)
1 - Mode on indicator (or enable)
RESET
(USB Reset)
0 - No reset
1 - Reset USB

Figure 26-35. USB Power Management Register

ENABLE_SUSPENDM

The ENABLE_SUSPENDM (bit 0) is set by the processor core to enable the SUSPENDM output (internal USB controller signal). When this bit is set, the SUSPENDM output signal is used by the USB PHY to power-down its drivers when the USB controller is not active.

SUSPEND_MODE

In host mode, SUSPEND_MODE (bit 1) is set by the processor core to enter suspend mode. In peripheral mode, this bit is set on entry into suspend

mode. It is cleared when the processor core reads the interrupt register, or sets the resume bit.

RESUME_MODE

The RESUME_MODE (bit 2) is set by the processor core to generate resume signaling when the function is in suspend mode. The processor core should clear this bit after 10 ms (a maximum of 15 ms) to end resume signaling. In host mode, this bit is also automatically set when resume signaling from the target is detected while the USB controller is suspended.

RESET

The RESET (bit 3) bit is set when reset signaling is present on the bus. This bit is read/write from the processor core in host mode but read-only in peripheral mode.

HS_MODE

When HS_MODE (bit 4) is set, this read-only bit indicates high-speed mode successfully negotiated during a USB reset. In peripheral mode, it becomes valid when the USB reset completes (as indicated by the USB reset interrupt). In host mode, it becomes valid when the RESET_OR_BABLE_B bit is cleared. It remains valid for the duration of the session.

HS_ENABLE

When HS_ENABLE (bit 5) is set by the processor core, the USB controller negotiates for high speed when the device is reset by the hub/host. If it is not set, the controller only operates in full-speed mode. By default HS_ENABLE is set to 1.

SOFT_CONN

If the soft connect/disconnect feature is enabled (bit 6, SOFT_CONN = 1), then the USB D+/D–lines are enabled when this bit is set by the processor

core and three-stated when this bit is cleared by the processor core. Only valid in peripheral mode.

ISO_UPDATE

When ISO_UPDATE (bit 7) is set by the processor core, the USB controller waits for an SOF token from the time TXPKTRDY is set before sending the packet. If an IN token is received before an SOF token, then a zero length data packet is sent. Only valid in peripheral mode. Also, this bit only affects endpoints performing isochronous transfers.

USB Function Address (USB_FADDR) Register

The USB_FADDR register (see Figure 26-36) contains the 7-bit address of the peripheral part of the transaction.

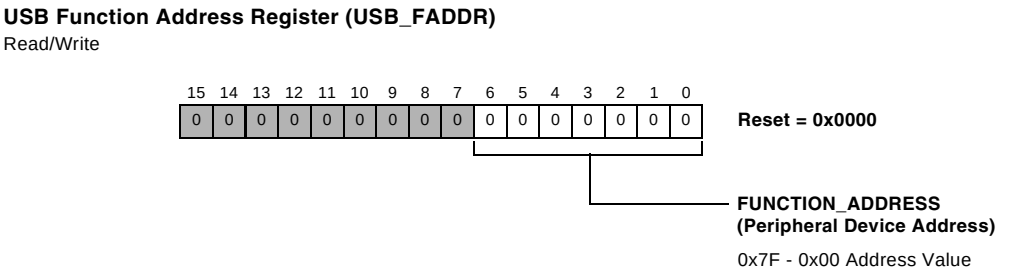

USB Function Address Register (USB_FADDR)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
FUNCTION_ADDRESS
(Peripheral Device Address)
0x7F - 0x00 Address Value

Figure 26-36. USB Function Address Register

When the USB controller is being used in peripheral mode (HOST_MODE = 0 in USB_OTG_DEV_CTL), this register is written with the address received through a SET_ADDRESS command. The address is used for decoding the function address in subsequent token packets.

When the USB controller is being used in host mode (HOST_MODE = 1 in USB_OTG_DEV_CTL), this register is set to the value sent in a SET_ADDRESS command during device enumeration as the address for the peripheral device.

USB Test Mode (USB_TESTMODE) Register

The USB_TESTMODE register (see Figure 26-37) places the USB controller into test mode state and also can put the USB controller into one of the four test modes for high-speed operation (see the USB 2.0 specification).

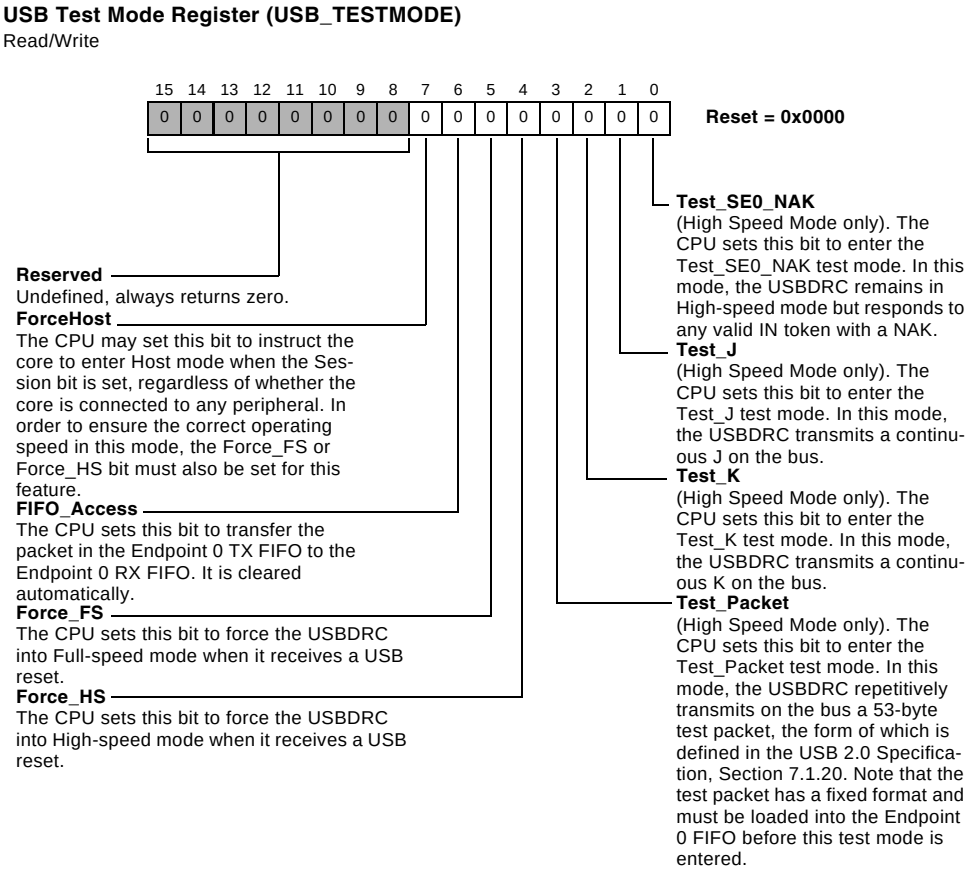

USB Test Mode Register (USB_TESTMODE)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Reserved
Undefined, always returns zero.
ForceHost
The CPU may set this bit to instruct the core to enter Host mode when the Session bit is set, regardless of whether the core is connected to any peripheral. In order to ensure the correct operating speed in this mode, the Force_FS or Force_HS bit must also be set for this feature.
FIFO_Access
The CPU sets this bit to transfer the packet in the Endpoint 0 TX FIFO to the Endpoint 0 RX FIFO. It is cleared automatically.
Force_FS
The CPU sets this bit to force the USBDRC into Full-speed mode when it receives a USB reset.
Force_HS
The CPU sets this bit to force the USBDRC into High-speed mode when it receives a USB reset.
Test_SE0_NAK
(High Speed Mode only). The CPU sets this bit to enter the Test_SE0_NAK test mode. In this mode, the USBDRC remains in High-speed mode but responds to any valid IN token with a NAK.
Test_J
(High Speed Mode only). The CPU sets this bit to enter the Test_J test mode. In this mode, the USBDRC transmits a continuous J on the bus.
Test_K
(High Speed Mode only). The CPU sets this bit to enter the Test_K test mode. In this mode, the USBDRC transmits a continuous K on the bus.
Test_Packet
(High Speed Mode only). The CPU sets this bit to enter the Test_Packet test mode. In this mode, the USBDRC repetitively transmits on the bus a 53-byte test packet, the form of which is defined in the USB 2.0 Specification, Section 7.1.20. Note that the test packet has a fixed format and must be loaded into the Endpoint 0 FIFO before this test mode is entered.

Figure 26-37. USB Test Mode Register

USB_TESTMODE is not used in normal operation. Only one of the bits may be set at one time, except for bit 5 in conjunction with the ForceHost feature.

USB Global Interrupt (USB_GLOBINTR) Register

The USB_GLOBINTR register (see Figure 26-38) selects routing for each of the three USB interrupt sources (USB_INTRRX, USB_INTRTX and USB_INTRUSB/USB_OTG_VBUS_IRQ) to any or all of the top-level interrupts (USB_INT0, USB_INT1 and USB_INT2).

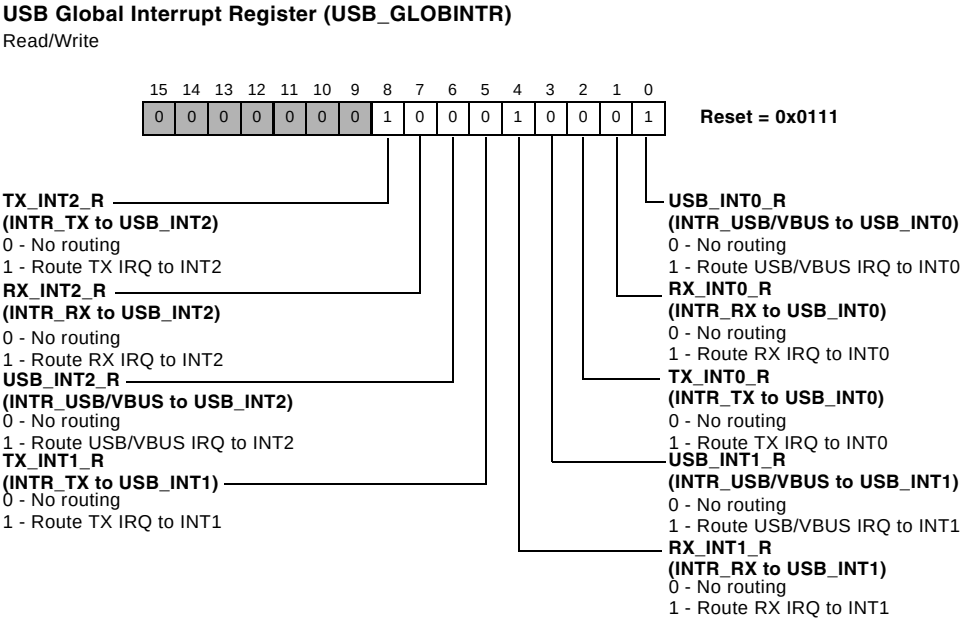

USB Global Interrupt Register (USB_GLOBINTR)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1
Reset = 0x0111
TX_INT2_R
(INTR_TX to USB_INT2)
0 - No routing
1 - Route TX IRQ to INT2
RX_INT2_R
(INTR_RX to USB_INT2)
0 - No routing
1 - Route RX IRQ to INT2
USB_INT2_R
(INTR_USB/VBUS to USB_INT2)
0 - No routing
1 - Route USB/VBUS IRQ to INT2
TX_INT1_R
(INTR_TX to USB_INT1)
0 - No routing
1 - Route TX IRQ to INT1
USB_INT0_R
(INTR_USB/VBUS to USB_INT0)
0 - No routing
1 - Route USB/VBUS IRQ to INT0
RX_INT0_R
(INTR_RX to USB_INT0)
0 - No routing
1 - Route RX IRQ to INT0
TX_INT0_R
(INTR_TX to USB_INT0)
0 - No routing
1 - Route TX IRQ to INT0
USB_INT1_R
(INTR_USB/VBUS to USB_INT1)
0 - No routing
1 - Route USB/VBUS IRQ to INT1
RX_INT1_R
(INTR_RX to USB_INT1)
0 - No routing
1 - Route RX IRQ to INT1

Figure 26-38. USB Global Interrupt Register

Each interrupt source is represented by a configuration bit across each of the top-level interrupts. Setting each to a 1, routes that source to the interrupt.

USB Transmit Interrupt (USB_INTRTX) Register

The USB_INTRTX register (see Figure 26-39) indicates which interrupts are currently active for endpoint 0 and the TX endpoints 1–7. Writing 1 to bits 0–7 when they are high clears that particular bit and de-asserts the corresponding interrupt source.

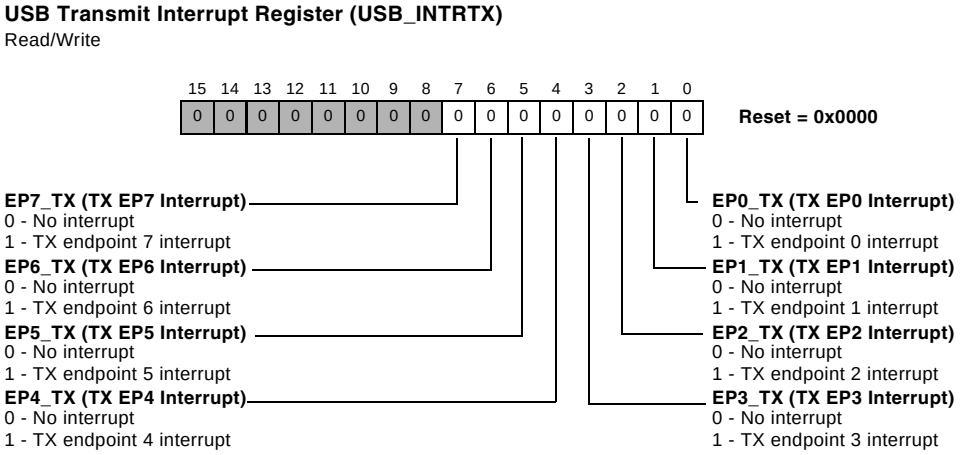

USB Transmit Interrupt Register (USB_INTRTX)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
EP7_TX (TX EP7 Interrupt)
0 - No interrupt
1 - TX endpoint 7 interrupt
EP6_TX (TX EP6 Interrupt)
0 - No interrupt
1 - TX endpoint 6 interrupt
EP5_TX (TX EP5 Interrupt)
0 - No interrupt
1 - TX endpoint 5 interrupt
EP4_TX (TX EP4 Interrupt)
0 - No interrupt
1 - TX endpoint 4 interrupt
EP0_TX (TX EP0 Interrupt)
0 - No interrupt
1 - TX endpoint 0 interrupt
EP1_TX (TX EP1 Interrupt)
0 - No interrupt
1 - TX endpoint 1 interrupt
EP2_TX (TX EP2 Interrupt)
0 - No interrupt
1 - TX endpoint 2 interrupt
EP3_TX (TX EP3 Interrupt)
0 - No interrupt
1 - TX endpoint 3 interrupt

Figure 26-39. USB Transmit Interrupt Register

USB Receive Interrupt (USB_INTRRX) Register

The USB_INTRRX register (see Figure 26-40) indicates which interrupts are currently active for the RX endpoints 1–7. Writing 1 to bits 1–7 when

they are high clears that particular bit and de-asserts the corresponding interrupt source.

USB Receive Interrupt Register (USB_INTRRX)

Read/Write

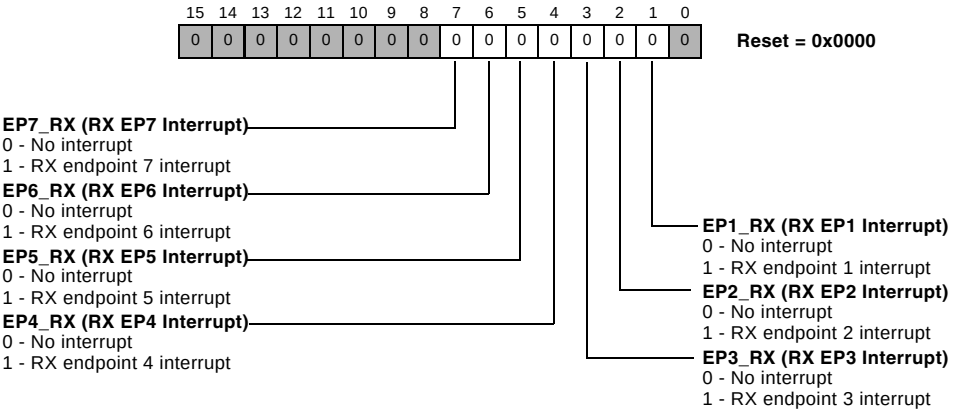

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
EP7_RX (RX EP7 Interrupt)
0 - No interrupt
1 - RX endpoint 7 interrupt
EP6_RX (RX EP6 Interrupt)
0 - No interrupt
1 - RX endpoint 6 interrupt
EP5_RX (RX EP5 Interrupt)
0 - No interrupt
1 - RX endpoint 5 interrupt
EP4_RX (RX EP4 Interrupt)
0 - No interrupt
1 - RX endpoint 4 interrupt
EP1_RX (RX EP1 Interrupt)
0 - No interrupt
1 - RX endpoint 1 interrupt
EP2_RX (RX EP2 Interrupt)
0 - No interrupt
1 - RX endpoint 2 interrupt
EP3_RX (RX EP3 Interrupt)
0 - No interrupt
1 - RX endpoint 3 interrupt

Figure 26-40. USB Receive Interrupt Register

USB Transmit Interrupt Enable (USB_INTRTXE) Register

The USB_INTRTXE register (see Figure 26-41) enables interrupts for endpoint 0 and the TX endpoints 1–7.

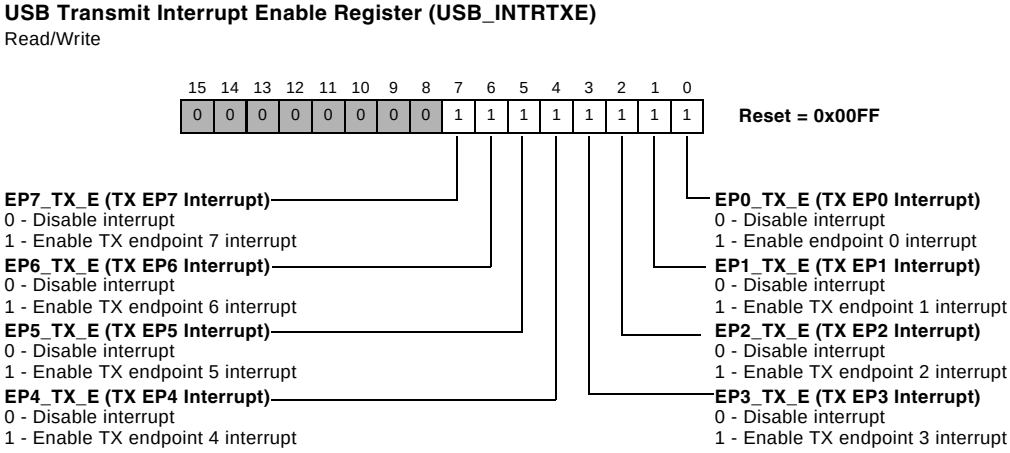

USB Transmit Interrupt Enable Register (USB_INTRTXE)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
Reset = 0x00FF
EP7_TX_E (TX EP7 Interrupt)
0 - Disable interrupt
1 - Enable TX endpoint 7 interrupt
EP6_TX_E (TX EP6 Interrupt)
0 - Disable interrupt
1 - Enable TX endpoint 6 interrupt
EP5_TX_E (TX EP5 Interrupt)
0 - Disable interrupt
1 - Enable TX endpoint 5 interrupt
EP4_TX_E (TX EP4 Interrupt)
0 - Disable interrupt
1 - Enable TX endpoint 4 interrupt
EP0_TX_E (TX EP0 Interrupt)
0 - Disable interrupt
1 - Enable endpoint 0 interrupt
EP1_TX_E (TX EP1 Interrupt)
0 - Disable interrupt
1 - Enable TX endpoint 1 interrupt
EP2_TX_E (TX EP2 Interrupt)
0 - Disable interrupt
1 - Enable TX endpoint 2 interrupt
EP3_TX_E (TX EP3 Interrupt)
0 - Disable interrupt
1 - Enable TX endpoint 3 interrupt

Figure 26-41. USB Transmit Interrupt Enable Register

Writing 1 to bits 0–7 enables (unmasks) the corresponding interrupt source. Writing 0 to bits 0–7 disables (masks) an interrupt source. The corresponding status bit in the USB_INTRTX register may still be set, but no interrupt is asserted. On reset, the bits corresponding to endpoint 0 and the TX endpoints included in the design are set to 1 (for example, all TX interrupts are enabled).

USB Receive Interrupt Enable (USB_INTRRXE) Register

The USB_INTRRXE register (see Figure 26-42) enables interrupts for the RX endpoints 1–7.

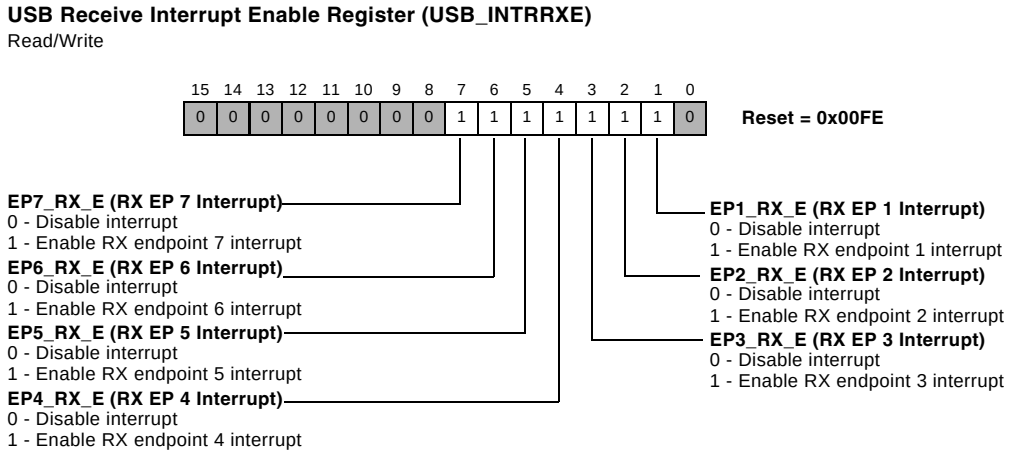

USB Receive Interrupt Enable Register (USB_INTRRXE)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0
Reset = 0x00FE
EP7_RX_E (RX EP 7 Interrupt)
0 - Disable interrupt
1 - Enable RX endpoint 7 interrupt
EP6_RX_E (RX EP 6 Interrupt)
0 - Disable interrupt
1 - Enable RX endpoint 6 interrupt
EP5_RX_E (RX EP 5 Interrupt)
0 - Disable interrupt
1 - Enable RX endpoint 5 interrupt
EP4_RX_E (RX EP 4 Interrupt)
0 - Disable interrupt
1 - Enable RX endpoint 4 interrupt
EP1_RX_E (RX EP 1 Interrupt)
0 - Disable interrupt
1 - Enable RX endpoint 1 interrupt
EP2_RX_E (RX EP 2 Interrupt)
0 - Disable interrupt
1 - Enable RX endpoint 2 interrupt
EP3_RX_E (RX EP 3 Interrupt)
0 - Disable interrupt
1 - Enable RX endpoint 3 interrupt

Figure 26-42. USB Receive Interrupt Enable Register

Writing 1 to bits 1–7 enables (unmasks) the corresponding interrupt source. Writing 0 to bits 1–7 disables (masks) an interrupt source. The corresponding status bit in the USB_INTRRX register may still be set, but no interrupt is asserted. On reset, the bits corresponding to endpoint 0 and the TX endpoints included in the design are set to 1 (for example, all TX interrupts are enabled).

USB Common Interrupts (USB_INTRUSB) Register

The USB_INTRUSB register (see Figure 26-43) indicates which USB interrupts are currently active.

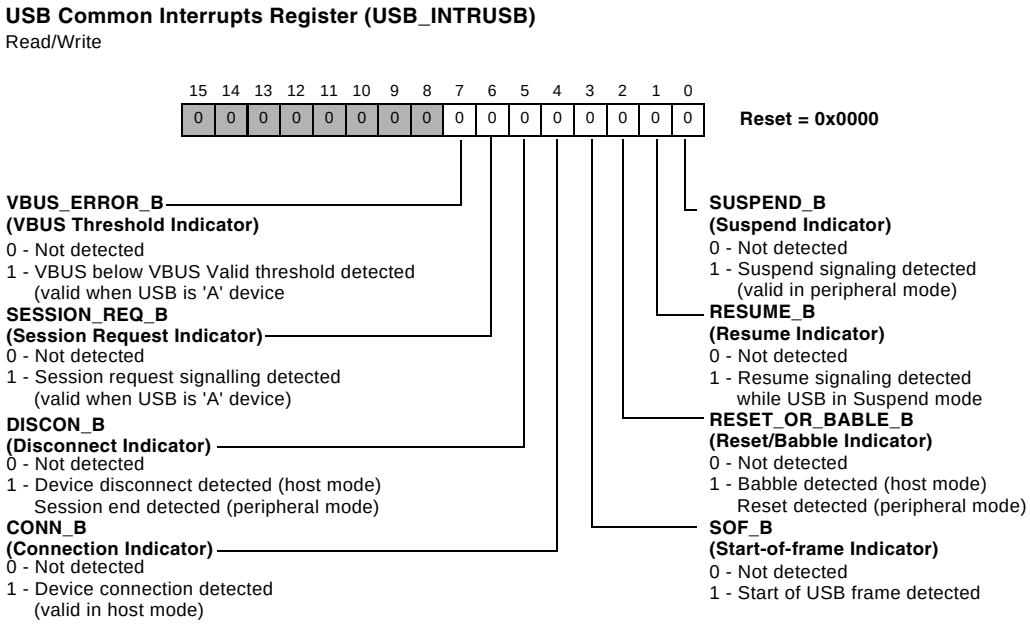

USB Common Interrupts Register (USB_INTRUSB)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
VBUS_ERROR_B
(VBUS Threshold Indicator)
0 - Not detected
1 - VBUS below VBUS Valid threshold detected
(valid when USB is 'A' device
SESSION_REQ_B
(Session Request Indicator)
0 - Not detected
1 - Session request signalling detected
(valid when USB is 'A' device)
DISCON_B
(Disconnect Indicator)
0 - Not detected
1 - Device disconnect detected (host mode)
Session end detected (peripheral mode)
CONN_B
(Connection Indicator)
0 - Not detected
1 - Device connection detected
(valid in host mode)
SUSPEND_B
(Suspend Indicator)
0 - Not detected
1 - Suspend signaling detected
(valid in peripheral mode)
RESUME_B
(Resume Indicator)
0 - Not detected
1 - Resume signaling detected
while USB in Suspend mode
RESET_OR_BABLE_B
(Reset/Babble Indicator)
0 - Not detected
1 - Babble detected (host mode)
Reset detected (peripheral mode)
SOF_B
(Start-of-frame Indicator)
0 - Not detected
1 - Start of USB frame detected

Figure 26-43. USB Common Interrupts Register

Writing a 1 to any of the bits 0–7 when they are high de-asserts the interrupt source corresponding to that bit. The USB_INTRUSB register shares an interrupt source line with USB_OTG_VBUS_IRQ.

USB Common Interrupt Enable (USB_INTRUSBE) Register

The USB_INTRUSBE register (see Figure 26-44) enables common USB interrupts. Writing 1 to bits 0–7 enables (unmasks) the corresponding interrupt source. Writing 0 to bits 0–7 disables (masks) an interrupt

source. The corresponding status bit in the USB_INTUSB register may still be set, but no interrupt is asserted. On reset, the RESUME_BE and RESET_OR_BABLE_BE bits are set to 1 (for example, interrupts for resume signalling detection and reset/babble detection are enabled).

USB Common Interrupts Enable Register (USB_INTRUSBE)

Read/Write

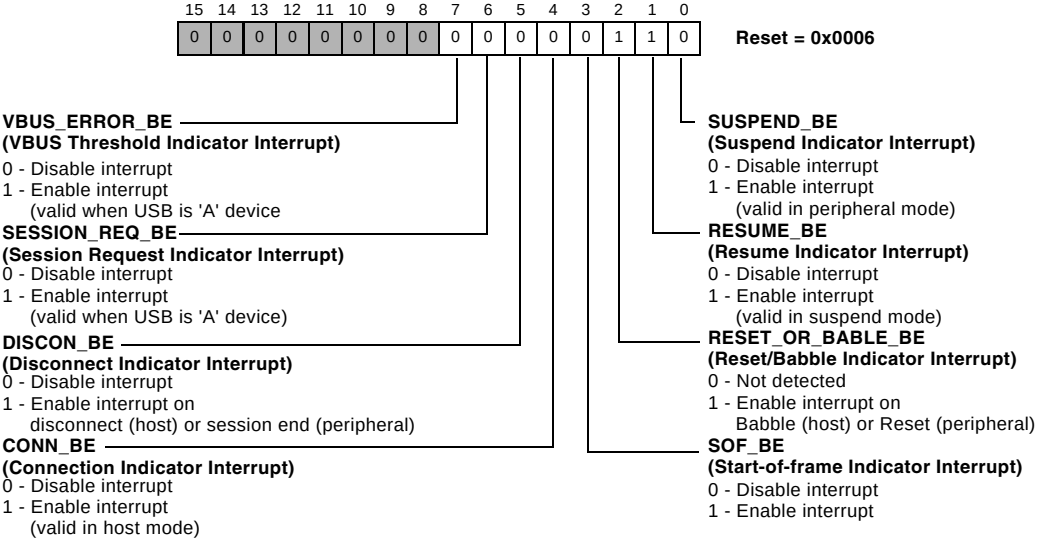

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
Reset = 0x0006
VBUS_ERROR_BE
(VBUS Threshold Indicator Interrupt)
0 - Disable interrupt
1 - Enable interrupt
(valid when USB is 'A' device
SESSION_REQ_BE
(Session Request Indicator Interrupt)
0 - Disable interrupt
1 - Enable interrupt
(valid when USB is 'A' device)
DISCON_BE
(Disconnect Indicator Interrupt)
0 - Disable interrupt
1 - Enable interrupt on
disconnect (host) or session end (peripheral)
CONN_BE
(Connection Indicator Interrupt)
0 - Disable interrupt
1 - Enable interrupt
(valid in host mode)
SUSPEND_BE
(Suspend Indicator Interrupt)
0 - Disable interrupt
1 - Enable interrupt
(valid in peripheral mode)
RESUME_BE
(Resume Indicator Interrupt)
0 - Disable interrupt
1 - Enable interrupt
(valid in suspend mode)
RESET_OR_BABLE_BE
(Reset/Babble Indicator Interrupt)
0 - Not detected
1 - Enable interrupt on
Babble (host) or Reset (peripheral)
SOF_BE
(Start-of-frame Indicator Interrupt)
0 - Disable interrupt
1 - Enable interrupt

Figure 26-44. USB Common Interrupts Enable Register

USB Frame Number (USB_FRAME) Register

The USB_FRAME register (see Figure 26-45) contains the last received frame number; bit 10 is MSB; bit 0 is LSB.

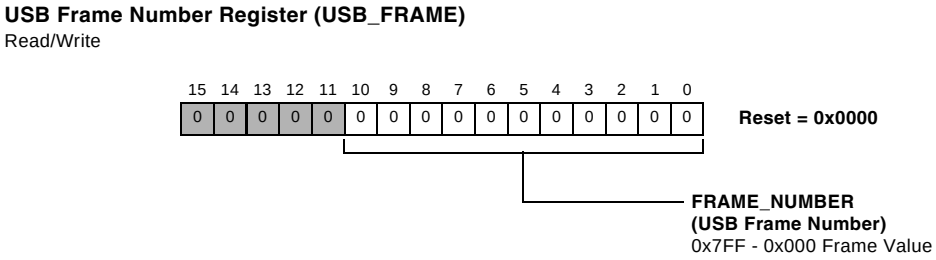

USB Frame Number Register (USB_FRAME)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
FRAME_NUMBER
(USB Frame Number)
0x7FF - 0x000 Frame Value

Figure 26-45. USB Frame Number Register

USB Index (USB_INDEX) Register

The USB_INDEX register (see Figure 26-46) contains an index value for alternate addressing of USB endpoint control and status registers.

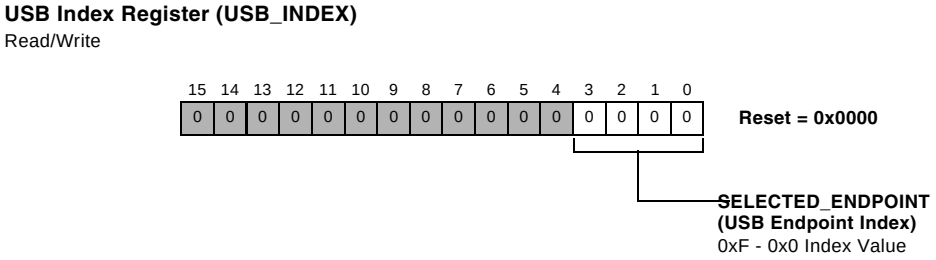

USB Index Register (USB_INDEX)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SELECTED_ENDPOINT
(USB Endpoint Index)
0xF - 0x0 Index Value

Figure 26-46. USB Index Register

Each TX endpoint and each RX endpoint have their own set of control/status registers located between address 0xFFC0 3E00 and 0xFFC0 3FF8. In addition, one indexed set of TX control/status and one set of RX control/status registers appear between address 0xFFC0 3C40 and 0xFFC0 3C68. The USB_INDEX is a 4-bit register that determines

which set of endpoint control/status registers are accessed at the indexed address range.

Before accessing an endpoint's control/status registers using the indexed range, the endpoint number is written to the USB_INDEX register to ensure that the correct control/status registers appear in the indexed range of the memory map.

USB TX Max Packet (USB_TX_MAX_PACKET) Register

The USB_TX_MAX_PACKET register (see Figure 26-47) defines the maximum amount of data that can be transferred through the selected transmit endpoint in a single frame. When setting this value, you must consider the constraints placed by the USB specification on packet sizes for bulk, interrupt and isochronous transactions in full-speed operations. The USB_TX_MAX_PACKET register provides indexed access to the USB_EP_NIx_TXMAXP register for each TX endpoint (except endpoint 0).

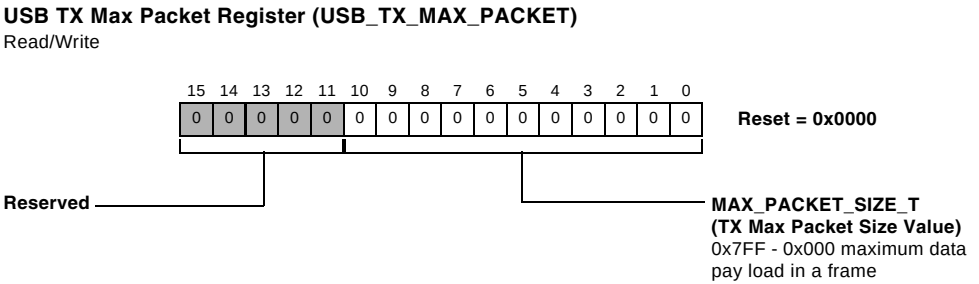

USB TX Max Packet Register (USB_TX_MAX_PACKET)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Reserved
MAX_PACKET_SIZE_T
(TX Max Packet Size Value)
0x7FF - 0x000 maximum data pay load in a frame

Figure 26-47. USB TX Max Packet Register

USB Control/Status EP0 (USB_CSR0) Register

The USB_CSR0 register (see Figure 26-48) provides control and status bits for endpoint 0. Note that some bits may be set to clear automatically. The

interpretation of the USB_CSR0 register depends on whether the USB controller is acting as a peripheral or as a host.

Many bits in this register have different operations (control versus status) depending on whether the USB is in peripheral or host mode. This register includes the following bits:

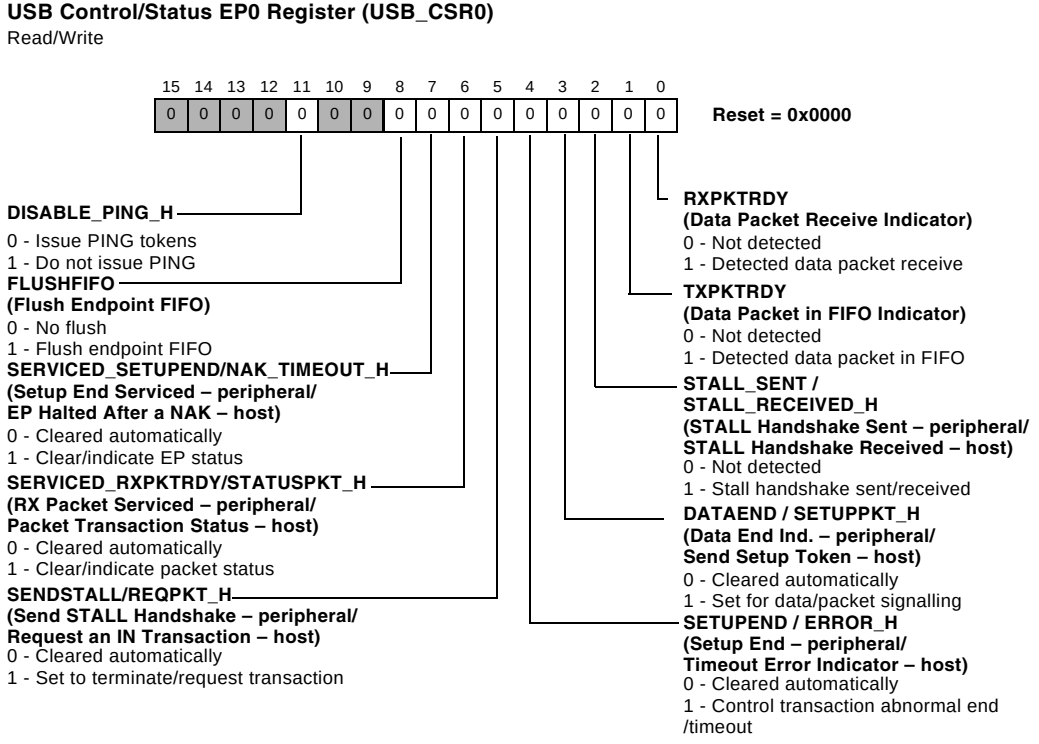

USB Control/Status EP0 Register (USB_CSR0)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DISABLE_PING_H
0 - Issue PING tokens
1 - Do not issue PING
FLUSHFIFO
(Flush Endpoint FIFO)
0 - No flush
1 - Flush endpoint FIFO
SERVICED_SETUPEND/NAK_TIMEOUT_H
(Setup End Serviced – peripheral/
EP Halted After a NAK – host)
0 - Cleared automatically
1 - Clear/indicate EP status
SERVICED_RXPKTRDY/STATUSPKT_H
(RX Packet Serviced – peripheral/
Packet Transaction Status – host)
0 - Cleared automatically
1 - Clear/indicate packet status
SENDSTALL/REQPKT_H
(Send STALL Handshake – peripheral/
Request an IN Transaction – host)
0 - Cleared automatically
1 - Set to terminate/request transaction
RXPKTRDY
(Data Packet Receive Indicator)
0 - Not detected
1 - Detected data packet receive
TXPKTRDY
(Data Packet in FIFO Indicator)
0 - Not detected
1 - Detected data packet in FIFO
STALL_SENT /
STALL_RECEIVED_H
(STALL Handshake Sent – peripheral/
STALL Handshake Received – host)
0 - Not detected
1 - Stall handshake sent/received
DATAEND / SETUPPKT_H
(Data End Ind. – peripheral/
Send Setup Token – host)
0 - Cleared automatically
1 - Set for data/packet signalling
SETUPEND / ERROR_H
(Setup End – peripheral/
Timeout Error Indicator – host)
0 - Cleared automatically
1 - Control transaction abnormal end
/timeout

Figure 26-48. USB Control/Status EP0 Register

RXPKTRDY

In peripheral mode, RXPKTRDY (bit 0) is set when a data packet is received. An interrupt is generated when this bit is set. The processor core clears this bit by setting the SERVICED_RXPKTRDY bit.

In host mode, RXPKTRDY (bit 0) is set when a data packet is received. An interrupt is generated (if enabled) when this bit is set. The processor core should clear this bit when the packet is read from the FIFO.

TXPKTRDY

In peripheral mode, the processor core sets TXPKTRDY (bit 1) after loading a data packet into the FIFO. It is cleared automatically when the data packet is transmitted. An interrupt is generated (if enabled) when the bit is cleared.

In host mode, the processor core sets TXPKTRDY (bit 1) after loading a data packet into the FIFO. It is cleared automatically when the data packet is transmitted. An interrupt is generated (if enabled) when the bit is cleared.

STALL_SENT / STALL_RECEIVED_H

In peripheral mode, STALL_SENT (bit 2) is set when a STALL handshake is transmitted. The processor core should clear this bit.

In host mode, STALL_RECEIVED_H (bit 2) is set when a STALL handshake is received. The processor core should clear this bit.

DATAEND / SETUPPKT_H

In peripheral mode, the processor core sets DATAEND (bit 3):

1. When setting TXPKTRDY for the last data packet.
2. When clearing RXPKTRDY after unloading the last data packet.
3. When setting TXPKTRDY for a zero length data packet. It is cleared automatically.

In host mode, the processor core sets SETUPPKT_H (bit 3), at the same time as the TXPKTRDY bit is set, to send a SETUP token instead of an OUT token for the transaction.

SETUPEND / ERROR_H

In peripheral mode, SETUPEND (bit 4) is set when a control transaction ends before the DATAEND bit is set. An interrupt is generated and the FIFO is flushed at this time. The bit is cleared by the processor core writing a 1 to the SERVICED_SETUPEND bit.

In host mode, ERROR_H (bit 4) is set when three attempts have been made to perform a transaction with no response from the peripheral. The processor core should clear this bit. An interrupt is generated when this bit is set.

SENDSTALL / REQPKT_H

In peripheral mode, the processor core writes a 1 to SENDSTALL (bit 5) to terminate the current transaction. The STALL handshake is transmitted, then this bit automatically is cleared.

In host mode, the processor core sets REQPKT_H (bit 5) to request an IN transaction. It is cleared when RXPKTRDY is set.

SERVICED_RXPKTRDY / STATUSPKT_H

In peripheral mode, the processor core writes a 1 to SERVICED_RXPKTRDY (bit 6) to clear the RXPKTRDY bit. It is cleared automatically.

In host mode, the processor core sets STATUSPKT_H (bit 6) at the same time as the TXPKTRDY or REQPKT_H bit is set, to perform a status stage transaction. Setting this bit ensures that the data toggle is set to 1 so that a DATA1 packet is used for the Status Stage transaction.

SERVICED_SETUPEND / NAK_TIMEOUT_H

In peripheral mode, the processor core writes a 1 to SERVICED_SETUPEND (bit 7) to clear the SETUPEND bit. It is cleared automatically.

In host mode, NAK_TIMEOUT_H (bit 7) is set when endpoint 0 is halted following the receipt of NAK responses for longer than the time set by the

USB_NAKLIMIT0 register. The processor core should clear this bit to allow the endpoint to continue.

FLUSHFIFO

In peripheral mode, the processor core writes a 1 to the FLUSHFIFO (bit 8) to flush the next packet to be transmitted/read from the endpoint 0 FIFO. The FIFO pointer is reset and the TXPKTRDY or RXPKTRDY bit (below) is cleared. FLUSHFIFO has no effect unless TXPKTRDY or RXPKTRDY is set.

In host mode, the processor core writes a 1 to FLUSHFIFO (bit 8) to flush the next packet to be transmitted/read from the endpoint 0 FIFO. The FIFO pointer is reset and the TXPKTRDY or RXPKTRDY bit (below) is cleared. FLUSHFIFO has no effect unless TXPKTRDY or RXPKTRDY is set.

DISABLE_PING_H

The processor core writes a 1 to this bit to instruct the USB controller not to issue PING tokens in data and status phases of a high-speed control transfer (for use with devices that do not respond to PING).

USB TX Control/Status EPx (USB_TXCSR) Register

The USB_TXCSR register (see Figure 26-49) provides control and status bits for transfers through the currently selected TX endpoint.

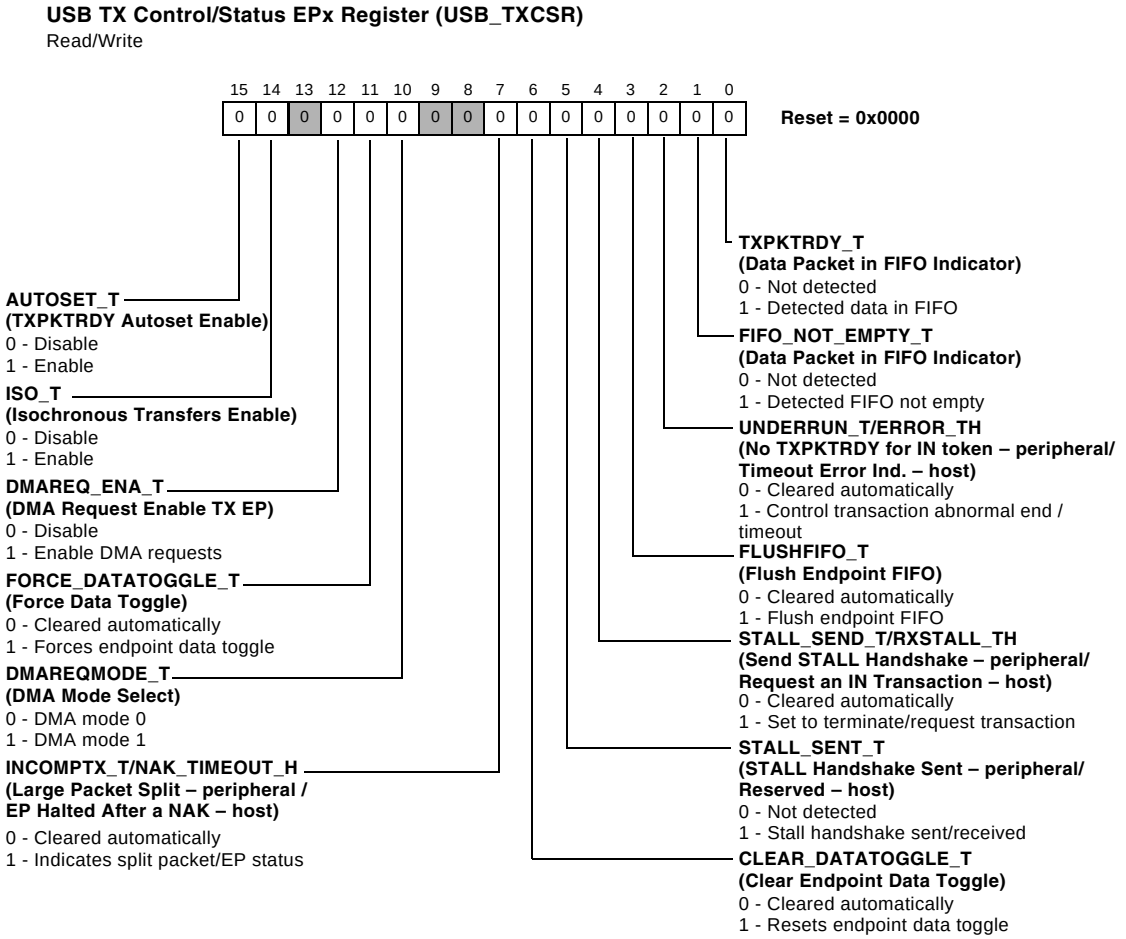

USB TX Control/Status EPx Register (USB_TXCSR)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
AUTOSET_T
(TXPKTRDY Autoset Enable)
0 - Disable
1 - Enable
ISO_T
(Isochronous Transfers Enable)
0 - Disable
1 - Enable
DMAREQ_ENA_T
(DMA Request Enable TX EP)
0 - Disable
1 - Enable DMA requests
FORCE_DATATOGGLE_T
(Force Data Toggle)
0 - Cleared automatically
1 - Forces endpoint data toggle
DMAREQMODE_T
(DMA Mode Select)
0 - DMA mode 0
1 - DMA mode 1
INCOMPTX_T/NAK_TIMEOUT_H
(Large Packet Split – peripheral /
EP Halted After a NAK – host)
0 - Cleared automatically
1 - Indicates split packet/EP status
TXPKTRDY_T
(Data Packet in FIFO Indicator)
0 - Not detected
1 - Detected data in FIFO
FIFO_NOT_EMPTY_T
(Data Packet in FIFO Indicator)
0 - Not detected
1 - Detected FIFO not empty
UNDERRUN_T/ERROR_TH
(No TXPKTRDY for IN token – peripheral/
Timeout Error Ind. – host)
0 - Cleared automatically
1 - Control transaction abnormal end / timeout
FLUSHFIFO_T
(Flush Endpoint FIFO)
0 - Cleared automatically
1 - Flush endpoint FIFO
STALL_SEND_T/RXSTALL_TH
(Send STALL Handshake – peripheral/
Request an IN Transaction – host)
0 - Cleared automatically
1 - Set to terminate/request transaction
STALL_SENT_T
(STALL Handshake Sent – peripheral/
Reserved – host)
0 - Not detected
1 - Stall handshake sent/received
CLEAR_DATATOGGLE_T
(Clear Endpoint Data Toggle)
0 - Cleared automatically
1 - Resets endpoint data toggle

Figure 26-49. USB TX Control/Status EPx Register

Note that some bits may be set to clear automatically. The interpretation of the USB_TXCSR register depends on whether the USB controller is acting as a peripheral or as a host.

There is a USB_EP_NIx_TXCSR register for each TX endpoint, except endpoint 0. These registers may be accessed directly through the register address or through the USB_TXCSR register indexed by the USB_INDEX register.

Many bits in the USB_TXCSR register have different operations (control versus status) depending on whether the USB is in peripheral or host mode. This register includes the following bits:

TXPKTRDY_T

In peripheral mode, the processor core sets TXPKTRDY_T (bit 0) after loading a data packet into the FIFO. It is cleared automatically when a data packet is transmitted. An interrupt is generated (if enabled) when the bit is cleared.

In host mode, the processor core sets TXPKTRDY_T (bit 0) after loading a data packet into the FIFO. It is cleared automatically when a data packet is transmitted. An interrupt is generated (if enabled) when the bit is cleared.

FIFO_NOT_EMPTY_T

In peripheral mode, the USB sets FIFO_NOT_EMPTY_T (bit 1) when there is at least one packet in the TX FIFO.

In host mode, the USB sets FIFO_NOT_EMPTY_T (bit 1) when there is at least one packet in the TX FIFO.

UNDERRUN_T / ERROR_TH

In peripheral mode, the USB sets UNDERRUN_T (bit 2) if an IN token is received when TXPKTRDY is not set. The processor core should clear this bit.

In host mode, the USB sets ERROR_TH (bit 2) when three attempts have been made to send a packet and no handshake packet is received. The processor core should clear this bit. An interrupt is generated when the bit is set. Valid only when the endpoint is operating in bulk or interrupt mode.

FLUSHFIFO_T

In peripheral mode, the processor core writes a 1 to FLUSHFIFO_T (bit 3) to flush the next packet to be transmitted from the endpoint TX FIFO. The FIFO pointer is reset and the TXPKTRDY bit (below) is cleared. This pointer may be set simultaneously with TxPktRdy to abort the packet that is currently being loaded into the FIFO. FLUSHFIFO_T has no effect unless TXPKTRDY is set. Note that if the FIFO is double-buffered, FLUSHFIFO_T may need to be set twice to completely clear the FIFO.

In host mode, the processor core writes a 1 to FLUSHFIFO_T (bit 3) to flush the next packet to be transmitted from the endpoint TX FIFO. The FIFO pointer is reset and the TXPKTRDY bit (below) is cleared. This pointer may be set simultaneously with TxPktRdy to abort the packet that is currently being loaded into the FIFO. FLUSHFIFO_T has no effect unless TXPKTRDY is set. Note that, if the FIFO is double-buffered, FLUSHFIFO_T may need to be set twice to completely clear the FIFO.

STALL_SEND_T / STALL_RECEIVED_TH

In peripheral mode, the processor core writes a 1 to STALL_SEND_T (bit 4) to issue a STALL handshake to an IN token. The processor core clears this bit to terminate the stall condition. This bit has no effect where the endpoint is being used for isochronous transfers.

In host mode, bit 4 is reserved.

STALL_SENT_T / RXSTALL_TH

In peripheral mode, SENTSTALL (bit 5) is set when a STALL handshake is transmitted. The FIFO is flushed and the TXPKTRDY bit is cleared. The processor core should clear this bit.

In host mode, RXSTALL_TH (bit 5) is set when a STALL handshake is received. The FIFO is flushed and the TXPKTRDY bit is cleared. The processor core should clear this bit.

CLEAR_DATATOGGLE_T

In peripheral mode, the processor core writes a 1 to CLEAR_DATATOGGLE_T (bit 6) to reset the endpoint data toggle to 0.

In host mode, the processor core writes a 1 to CLEAR_DATATOGGLE_T (bit 6) to reset the endpoint data toggle to 0.

INCOMPTX_T / NAK_TIMEOUT_TH

In peripheral mode, this bit always returns 0.

In host mode, NAK_TIMEOUT_TH (bit 7) is set when the TX endpoint is halted following the receipt of NAK responses for longer than the time set as the NAK limit by the USB_TXINTERVAL register. The processor core should clear this bit to allow the endpoint to continue. This bit is valid only for bulk endpoints.

DMAREQMODE_T

In peripheral mode, the processor core sets DMAREQMODE_T (bit 10) to select DMA mode 1 and clears this bit to select DMA mode 0.

In host mode, the processor core sets DMAREQMODE_T (bit 10) to select DMA mode 1 and clears this bit to select DMA mode 0.

FORCE_DATATOGGLE_T

In peripheral mode, the processor core sets FORCE_DATATOGGLE_T (bit 11) to force the endpoint data toggle to switch and the data packet to be cleared from the FIFO, regardless of whether an ACK was received. This can be used by interrupt TX endpoints that are used to communicate rate feedback for isochronous endpoints.

In host mode, the processor core sets FRCDATATOG (bit 11) to force the endpoint data toggle to switch and the data packet to be cleared from the FIFO, regardless of whether an ACK was received. This can be used by interrupt TX endpoints that are used to communicate rate feedback for isochronous endpoints.

DMAREQ_ENA_T

In peripheral mode, the processor core sets DMAREQ_ENA_T (bit 12) to enable the DMA request for the TX endpoint.

In host mode, the processor core sets DMAREQ_ENA_T (bit 12) to enable the DMA request for the TX endpoint.

ISO_T

In peripheral mode, the processor core sets ISO_T (bit 14) to enable the TX endpoint for isochronous transfers, and clears it to enable the TX endpoint for bulk or interrupt transfers. This bit only has an effect in peripheral mode.

In host mode, bit 14 is unused, and always returns zero.

AUTOSET_T

In peripheral mode, if the processor core sets AUTOSET_T (bit 15), TXPKTRDY automatically is set when data of the maximum packet size (value in USB_TX_MAX_PACKET) is loaded into the TX FIFO. If a packet of less than the maximum packet size is loaded, then TXPKTRDY must be set manually.

In host mode, if the processor core sets AUTOSET_T (bit 15), TXPKTRDY automatically is set when data of the maximum packet size (value in USB_TX_MAX_PACKET) is loaded into the TX FIFO. If a packet of less than the maximum packet size is loaded, then TXPKTRDY must be set manually.

USB RX Max Packet (USB_RX_MAX_PACKET) Register

The USB_RX_MAX_PACKET register (see Figure 26-50) defines the maximum amount of data that can be transferred through the selected transmit endpoint in a single frame.

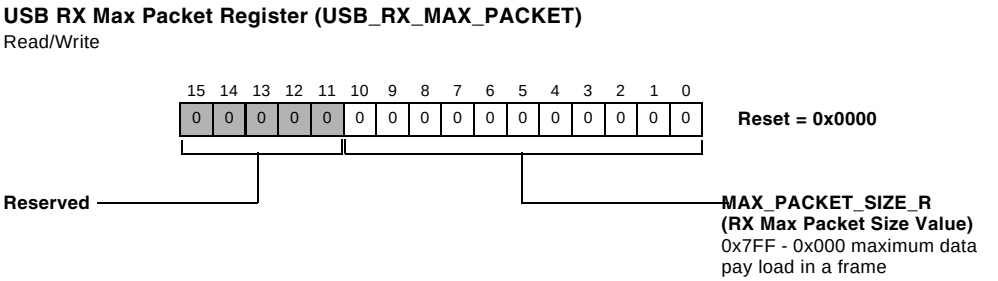

USB RX Max Packet Register (USB_RX_MAX_PACKET)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Reserved
MAX_PACKET_SIZE_R
(RX Max Packet Size Value)
0x7FF - 0x000 maximum data pay load in a frame

Figure 26-50. USB RX Max Packet Register

The USB_RX_MAX_PACKET register provides indexed access to the USB_EP_NIx_RXMAXP register for each RX endpoint (except endpoint 0). Bits[10:0] define (in bytes) the maximum payload transmitted in a single transaction. The legal value loaded can be up to 1023 bytes but is subject to the constraints placed by the USB specification on packet sizes for bulk, interrupt and isochronous transfers in full-speed operation.

A value greater than the maximum allowed of 1023 for full-speed USB operation produces unpredictable results.

The value written to this register should match the programmed maximum individual packet size (*MaxPktSize*) of the standard endpoint descriptor for the associated endpoint (see *Universal Serial Bus Specification Revision 2.0*, Chapter 9). A mismatch could cause unexpected results.

The total amount of data represented by the value written to this register must not exceed the RX FIFO size, and should not exceed half the FIFO size if double-buffering is required.

USB RX Control/Status (USB_RXCSR) Register

The USB_RXCSR register (see Figure 26-51) provides control and status bits for transfers through the currently selected RX endpoint.

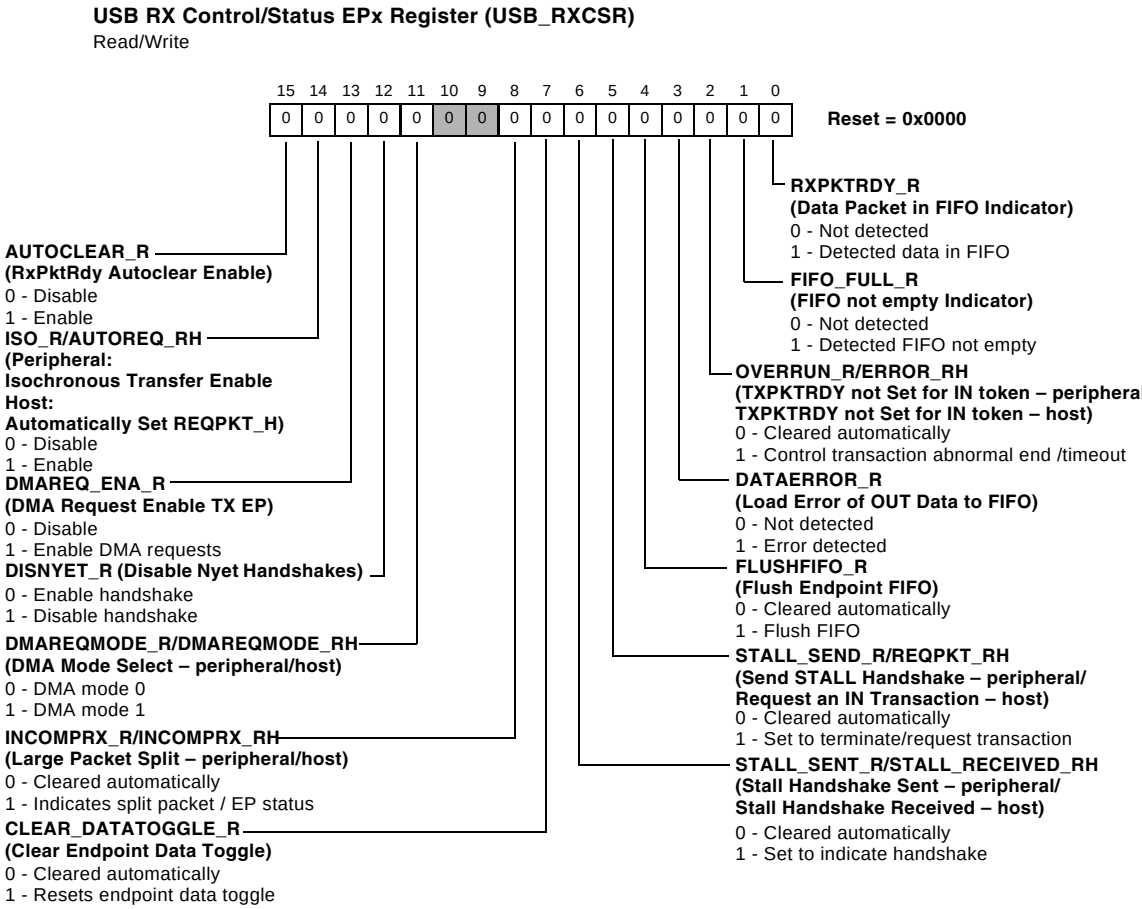

USB RX Control/Status EPx Register (USB_RXCSR)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RXPKTRDY_R
(Data Packet in FIFO Indicator)
0 - Not detected
1 - Detected data in FIFO
FIFO_FULL_R
(FIFO not empty Indicator)
0 - Not detected
1 - Detected FIFO not empty
OVERRUN_R/ERROR_RH
(TXPKTRDY not Set for IN token – peripheral
TXPKTRDY not Set for IN token – host)
0 - Cleared automatically
1 - Control transaction abnormal end /timeout
DATAERROR_R
(Load Error of OUT Data to FIFO)
0 - Not detected
1 - Error detected
FLUSHFIFO_R
(Flush Endpoint FIFO)
0 - Cleared automatically
1 - Flush FIFO
STALL_SEND_R/REQPKT_RH
(Send STALL Handshake – peripheral/
Request an IN Transaction – host)
0 - Cleared automatically
1 - Set to terminate/request transaction
STALL_SENT_R/STALL_RECEIVED_RH
(Stall Handshake Sent – peripheral/
Stall Handshake Received – host)
0 - Cleared automatically
1 - Set to indicate handshake
AUTOCLEAR_R
(RxPktRdy Autoclear Enable)
0 - Disable
1 - Enable
ISO_R/AUTOREQ_RH
(Peripheral:
Isochronous Transfer Enable
Host:
Automatically Set REQPKT_H)
0 - Disable
1 - Enable
DMAREQ_ENA_R
(DMA Request Enable TX EP)
0 - Disable
1 - Enable DMA requests
DISNYET_R (Disable Nyet Handshakes)
0 - Enable handshake
1 - Disable handshake
DMAREQMODE_R/DMAREQMODE_RH
(DMA Mode Select – peripheral/host)
0 - DMA mode 0
1 - DMA mode 1
INCOMPRX_R/INCOMPRX_RH
(Large Packet Split – peripheral/host)
0 - Cleared automatically
1 - Indicates split packet / EP status
CLEAR_DATATOGGLE_R
(Clear Endpoint Data Toggle)
0 - Cleared automatically
1 - Resets endpoint data toggle

Figure 26-51. USB RX Control/Status EPx Register

Note that some bits may be set to clear automatically. The interpretation of the USB_RXCSR register depends on whether the USB controller is acting as a peripheral or as a host.

There is a USB_EP_NIx_RXCSR register for each RX endpoint, except endpoint 0. These registers may be accessed directly through the register address or through the USB_RXCSR register indexed by the USB_INDEX register.

Many bits in the USB_RXCSR register have different operations (control versus status) depending on whether the USB is in peripheral or host mode. This register includes the following bits:

RXPKTRDY_R

In peripheral mode, RXPKTRDY_R (bit 0) is set when a data packet is received. The processor core should clear this bit when the packet is unloaded from the RX FIFO. An interrupt is generated when the bit is set.

In host mode, RXPKTRDY_R (bit 0) is set when a data packet is received. The processor core should clear this bit when the packet is unloaded from the RX FIFO. An interrupt is generated when the bit is set.

FIFO_FULL_R

In peripheral mode, FIFO_FULL_R (bit 1) is set when no more packets can be loaded into the RX FIFO.

In host mode, FIFO_FULL_R (bit 1) is set when no more packets can be loaded into the RX FIFO.

OVERRUN_R / ERROR_RH

In peripheral mode, OVERRUN_R (bit 2) is set if an OUT packet cannot be loaded into the RX FIFO. The processor core should clear this bit. This bit is only valid when the endpoint is operating in isochronous mode. In bulk mode, it always returns zero.

In host mode, the USB sets ERROR_RH (bit 2) when 3 attempts have been made to receive a packet and no data packet is received. The processor core should clear this bit. An interrupt is generated when the bit is set. This bit is only valid when the TX endpoint is operating in bulk or interrupt mode. In isochronous mode, it always returns zero.

DATAERROR_R

In peripheral mode, DATAERROR_R (bit 3) is set when RXPKTRDY is set if the data packet has a CRC or bit-stuff error. It is cleared when RXPKTRDY is cleared. This bit is only valid when the endpoint is operating in isochronous mode. In bulk mode, it always returns zero.

In host mode, when operating in isochronous mode, DATAERROR_R (bit 3) is set when RXPKTRDY is set and the data packet has a CRC or bit-stuff error and cleared when RXPKTRDY is cleared. In bulk mode, this bit is set when the RX endpoint is halted following the receipt of NAK responses for longer than the time set as the NAK limit by the USB_RXINTERVAL register. The processor core should clear this bit to allow the endpoint to continue.

FLUSHFIFO_R

In peripheral mode, the processor core writes a 1 to FLUSHFIFO_R (bit 4) to flush the next packet to be read from the endpoint RX FIFO. The FIFO pointer is reset and the RXPKTRDY bit (below) is cleared. FLUSHFIFO_R has no effect unless RXPKTRDY is set. Note that, if the FIFO is double-buffered, FLUSHFIFO_R may need to be set twice to completely clear the FIFO.

In host mode, the processor core writes a 1 to FLUSHFIFO_R (bit 4) to flush the next packet to be read from the endpoint RX FIFO. The FIFO pointer is reset and the RXPKTRDY bit (below) is cleared. FLUSHFIFO_R has no effect unless RXPKTRDY is set. Note that, if the FIFO is double-buffered, FLUSHFIFO_R may need to be set twice to completely clear the FIFO.

STALL_SEND_R / REQPKT_RH

In peripheral mode, the processor core writes a 1 to STALL_SEND_R (bit 5) to issue a STALL handshake. The processor core clears this bit to terminate the stall condition. This bit has no effect where the endpoint is being used for isochronous transfers.

In host mode, the processor core writes a 1 to REQPKT_RH (bit 5) to request an IN transaction. It is cleared when RXPKTRDY is set.

STALL_SENT_R / STALL_RECEIVED_RH

In peripheral mode, STALL_SENT_R (bit 6) is set when a STALL handshake is transmitted. The processor core should clear this bit.

In host mode, when a STALL handshake is received, STALL_RECEIVED_RH (bit 6) is set and an interrupt is generated. The processor core should clear this bit.

CLEAR_DATATOGGLE_R

In peripheral mode, the processor core writes a 1 to CLEAR_DATATOGGLE_R (bit 7) to reset the endpoint data toggle to 0.

In host mode, the processor core writes a 1 to CLEAR_DATATOGGLE_R (bit 7) to reset the endpoint data toggle to 0.

INCOMPRX_R / INCOMPRX_RH

In peripheral mode, INCOMPRX_R (bit 8) always returns 0.

In host mode, INCOMPRX_RH (bit 8) always returns 0.

DMAREQMODE_R / DMAREQMODE_RH

In peripheral mode, the processor core sets DMAREQMODE_R (bit 11) to select DMA request mode 1 and clears this bit to select DMA request mode 0.

In host mode, the processor core sets DMAREQMODE_RH (bit 11) to select DMA mode 1 and clears this bit to select DMA mode 0.

DISNYET_R

In peripheral mode, the processor core sets DISNYET_R (bit 12) to disable the sending of NYET handshakes.When set, all successfully received RX packets are acknowledged, including at the point at which the FIFO becomes full. This bit only has an effect in high-speed mode, where it is set for all interrupt endpoints.

In host mode, the processor core sets DISNYET_R (bit 12) to disable the sending of NYET handshakes.When set, all successfully received RX packets are acknowledged including the point at which the FIFO becomes full. This bit only has an effect in high-speed mode, where it is set for all interrupt transfers.

DMAREQ_ENA_R

In peripheral mode, the processor core sets DMAREQ_ENA_R (bit 13) to enable the DMA request for the RX endpoint.

In host mode, the processor core sets DMAREQ_ENA_R (bit 13) to enable the DMA request for the RX endpoint.

ISO_R / AUTOREQ_RH

In peripheral mode, the processor core sets ISO_R (bit 14) to enable the RX endpoint for isochronous transfers, and clears it to enable the RX endpoint for bulk or interrupt transfers.

In host mode, if the processor core sets AUTOREQ_RH (bit 14), the REQPKT_H bit automatically is set when the RXPKTRDY bit is cleared.

AUTOCLEAR_R

In peripheral mode, if the processor core sets AUTOCLEAR_R (bit 15), the RXPKTRDY bit automatically is cleared when a packet of USB_RX_MAX_PACKET

bytes is unloaded from the RX FIFO. When packets of less than the maximum packet size are unloaded, RXPKTRDY must be cleared manually.

In host mode, if the processor core sets AUTOCLEAR_R (bit 15), the RXPKTRDY bit automatically is cleared when a packet of USB_RX_MAX_PACKET bytes is unloaded from the RX FIFO. When packets of less than the maximum packet size are unloaded, RXPKTRDY must be cleared manually.

USB Count 0 (USB_COUNT0) Register

The USB_COUNT0 register (see Figure 26-52) indicates the number of received data bytes in the endpoint 0 FIFO. The value returned changes as the contents of the FIFO change and is only valid while RXPKTRDY is set.

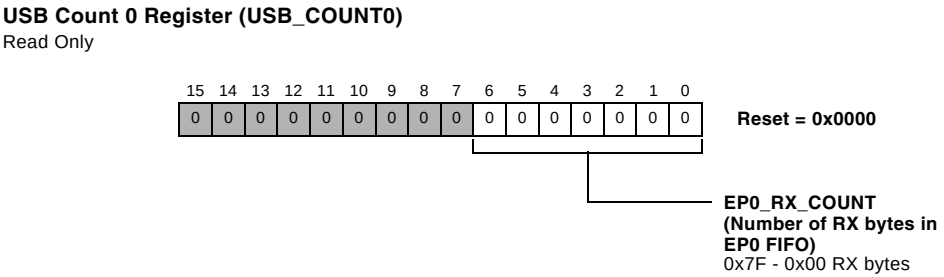

USB Count 0 Register (USB_COUNT0)
Read Only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
EP0_RX_COUNT
(Number of RX bytes in
EP0 FIFO)
0x7F - 0x00 RX bytes

Figure 26-52. USB Count 0 Register

USB RX Byte Count EPx (USB_RXCOUNT) Register

The USB_RXCOUNT register (see Figure 26-53) holds the number of received data bytes in the packet in the RX FIFO. Note that the value returned

changes as the FIFO is unloaded and is only valid while RXPKTRDY in USB_RXCSR is set.

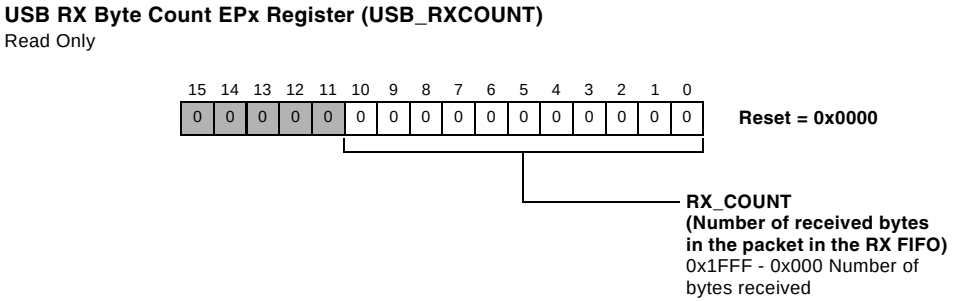

USB RX Byte Count EPx Register (USB_RXCOUNT)
Read Only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RX_COUNT
(Number of received bytes
in the packet in the RX FIFO)
0x1FFF - 0x000 Number of
bytes received

Figure 26-53. USB RX Byte Count Register

USB TX Type (USB_TXTYPE) Register

The USB_TXTYPE register (see Figure 26-54) selects the endpoint number and transaction protocol to use for the currently selected TX endpoint. There is a USB_TXTYPE register for each TX endpoint.

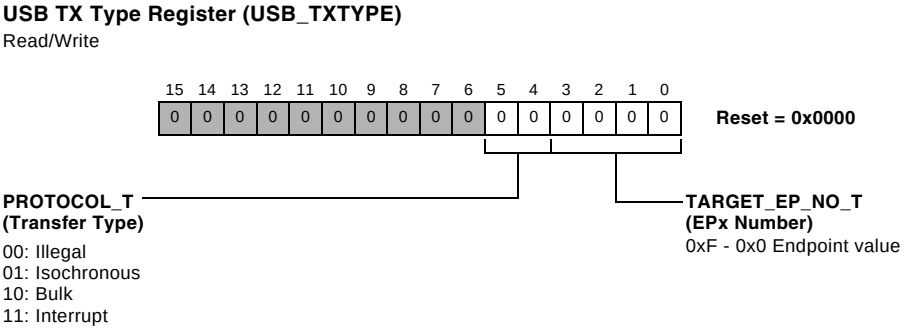

USB TX Type Register (USB_TXTYPE)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
PROTOCOL_T
(Transfer Type)
00: Illegal
01: Isochronous
10: Bulk
11: Interrupt
TARGET_EP_NO_T
(EPx Number)
0xF - 0x0 Endpoint value

Figure 26-54. USB TX Type Register

USB NAK Limit 0 (USB_NAKLIMIT0) Register

The USB_NAKLIMIT0 register (see Figure 26-55) determines the number of frames/micro-frames after which the endpoint should timeout on receiving a stream of NAK responses for bulk endpoints.

USB NAK Limit 0 Register (USB_NAKLIMIT0)
Read Only

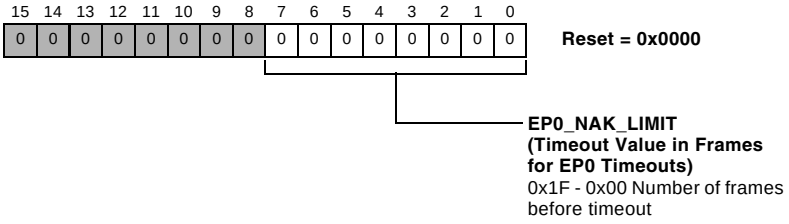

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
EP0_NAK_LIMIT
(Timeout Value in Frames
for EP0 Timeouts)
0x1F - 0x00 Number of frames
before timeout

Figure 26-55. USB NAK Limit 0 Register

USB TX Interval (USB_TXINTERVAL) Register

The USB_TXINTERVAL register (see Figure 26-56) defines the polling interval for the currently selected TX endpoint for interrupt, isochronous, and bulk transfers. There is a USB_TXINTERVAL register for each configured TX endpoint, *except* endpoint 0

USB TX Interval Register (USB_TXINTERVAL)
Read/Write

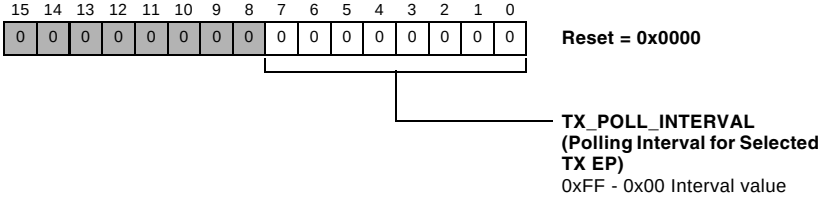

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
TX_POLL_INTERVAL
(Polling Interval for Selected
TX EP)
0xFF - 0x00 Interval value

Figure 26-56. USB TX Interval Register

Table 26-4 relates transfer types to TX_POLL_INTERVAL values (number of frames).

Table 26-3. Interval Value Versus Transfer Type

Transfer Type	Speed	Valid Values (m)	Interpretation
Interrupt	Low Speed or Full Speed	1 – 255	Polling interval is m frames.
	High Speed	1 – 16	Polling interval is $2^{(m-1)}$ micro-frames.
Isochronous	Full Speed or High Speed	1 – 16	Polling interval is $2^{(m-1)}$ frames or micro-frames.
Bulk	Full Speed or High Speed	2 – 16	NAK Limit is $2^{(m-1)}$ frames or micro-fames. Note: A value of 0 or 1 disables the NAK timeout function.

USB RX Type (USB_RXTYPE) Register

The USB_RXTYPE register (see Figure 26-57) selects the endpoint number and transaction protocol to use for the currently selected RX endpoint. There is a USB_RXTYPE register for each RX, *except* endpoint 0.

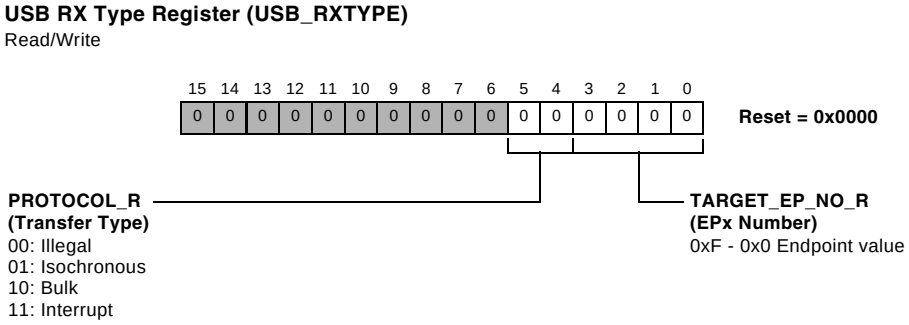

USB RX Type Register (USB_RXTYPE)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
PROTOCOL_R
(Transfer Type)
00: Illegal
01: Isochronous
10: Bulk
11: Interrupt
TARGET_EP_NO_R
(EPx Number)
0xF - 0x0 Endpoint value

Figure 26-57. USB RX Type Register

USB RX Interval (USB_RXINTERVAL) Register

The USB_RXINTERVAL register (see Figure 26-58) defines the polling interval in number of frames for the currently selected RX endpoint for interrupt, isochronous, and bulk transfers. There is a USB_RXINTERVAL register for each configured RX endpoint, *except* endpoint 0.

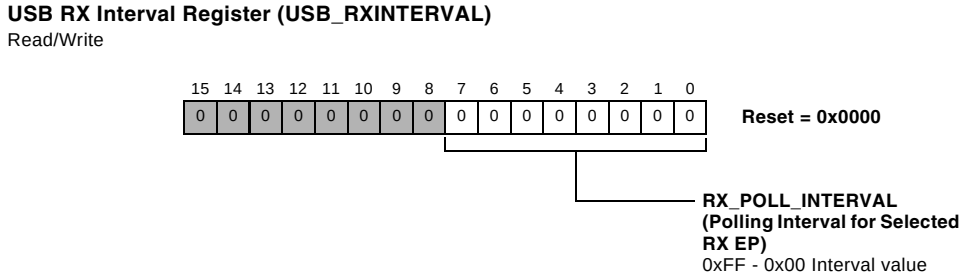

USB RX Interval Register (USB_RXINTERVAL)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RX_POLL_INTERVAL
(Polling Interval for Selected
RX EP)
0xFF - 0x00 Interval value

Figure 26-58. USB RX Interval Register

Table 26-4 relates transfer types to RX_POLL_INTERVAL values (number of frames).

Table 26-4. Interval Value Versus Transfer Type

Transfer Type	Speed	Valid Values (m)	Interpretation
Interrupt	Low Speed or Full Speed	1 – 255	Polling interval is m frames.
	High Speed	1 – 16	Polling interval is $2^{(m-1)}$ micro-frames.
Isochronous	Full Speed or High Speed	1 – 16	Polling interval is $2^{(m-1)}$ frames or micro-frames.
Bulk	Full Speed or High Speed	2 – 16	NAK Limit is $2^{(m-1)}$ frames or micro-fames. Note: A value of 0 or 1 disables the NAK timeout function.

USB TX Byte Count EPx (USB_TXCOUNT) Register

The USB_TXCOUNT register (see Figure 26-59) selects the size in bytes of the packet/transfer which is about to be written into a TX endpoint FIFO.

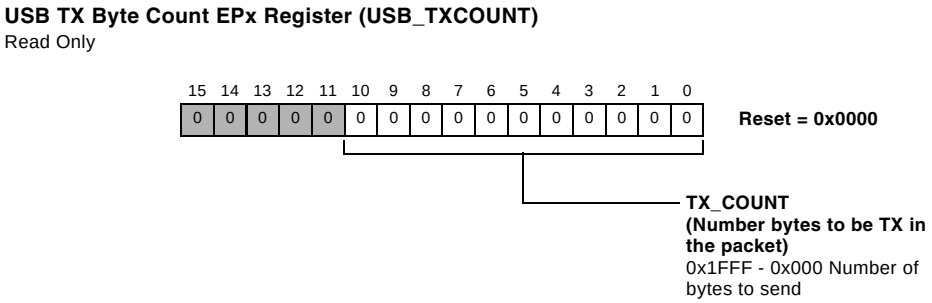

USB TX Byte Count EPx Register (USB_TXCOUNT)
Read Only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
TX_COUNT
(Number bytes to be TX in the packet)
0x1FFF - 0x000 Number of bytes to send

Figure 26-59. USB TX Byte Count EPx Register

As the packet is transferred, USB_TXCOUNT is a register that can be used by the processor core to program the size in bytes of the packet/transfer that is about to be written into a TX endpoint FIFO. The value is decremented by two when the processor core writes to the corresponding USB_EPx_FIFO high word address and is decremented by one when the processor core writes a byte to the FIFO using the corresponding USB_EPx_FIFO low word address. If the count itself reaches 0x0001 (which would only happen for odd-sized transfers), the next write into *either* USB_EPx_FIFO high word *or* USB_EPx_FIFO low word writes only the least significant byte of the half word into the FIFO. This aids DMA transfers that require IO accesses to go to the same address. USB_TXCOUNT must be re-loaded after it has counted to zero. It is not activated until it is loaded with a non-zero value.

See "Loading/Unloading Packets from Endpoints" on page 26-86 for more information about using USB_TXCOUNT.

USB Endpoint FIFO (USB_EPx_FIFO) Registers

Each endpoint uses a FIFO register (USB_EPx_FIFO) for data transfer. For more information about these FIFOs, see "Data Transfer" on page 26-85.

USB OTG Device Control (USB_OTG_DEV_CTL) Register

The USB_OTG_DEV_CTL register (see Figure 26-60) selects whether the USB controller is operating in peripheral mode or in host mode, and for controlling and monitoring the USB VBUS line.

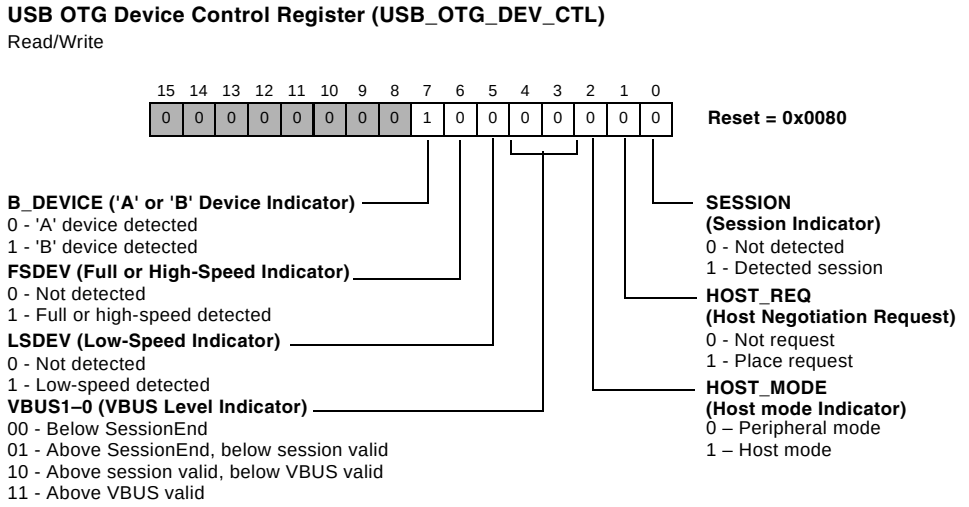

USB OTG Device Control Register (USB_OTG_DEV_CTL)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
Reset = 0x0080
B_DEVICE ('A' or 'B' Device Indicator)
0 - 'A' device detected
1 - 'B' device detected
FSDEV (Full or High-Speed Indicator)
0 - Not detected
1 - Full or high-speed detected
LSDEV (Low-Speed Indicator)
0 - Not detected
1 - Low-speed detected
VBUS1–0 (VBUS Level Indicator)
00 - Below SessionEnd
01 - Above SessionEnd, below session valid
10 - Above session valid, below VBUS valid
11 - Above VBUS valid
SESSION
(Session Indicator)
0 - Not detected
1 - Detected session
HOST_REQ
(Host Negotiation Request)
0 - Not request
1 - Place request
HOST_MODE
(Host mode Indicator)
0 – Peripheral mode
1 – Host mode

Figure 26-60. USB OTG Device Control Register

SESSION

When operating as an 'A' device, SESSION (bit 0) is set or cleared by the processor core to start or end a session. When operating as a 'B' device, SESSION is set/cleared by the USB controller when a session starts/ends. SESSION is also set by the processor core to initiate the session request pro-

tocol. When the USB controller is in Suspend mode, the bit may be cleared by the processor core to perform a software disconnect.

HOST_REQ

When HOST_REQ (bit 1) is set, the USB controller initiates the host negotiation when suspend mode is entered. HOST_REQ is cleared when host negotiation is completed. ('B' device only)

HOST_MODE

The HOST_MODE (bit 2) read-only bit is set when the USB controller is acting as a host.

VBUS0[1:0]

The VBUS (bits 4–3) bits are read-only bits that encode the current VBUS level.

LSDEV

The LSDEV (bit 5) read-only bit is set when a low-speed device is detected being connected to the port. Only valid in host mode.

FSDEV

The FSDEV (bit 6) read-only bit is set when a full-speed or high-speed device is detected being connected to the port. High speed devices are distinguished from full-speed by checking for high-speed chirps when the device detects a USB reset. Only valid in host mode.

B_DEVICE

The B_DEVICE (bit 7) read-only bit indicates whether the USB controller is operating as the 'A' device or the 'B' device. Only valid while a session is in progress.

USB OTG VBUS Interrupt (USB_OTG_VBUS_IRQ) Register

The USB_OTG_VBUS_IRQ register (see Figure 26-61) is an interrupt status register used to indicate when VBUS is required to be driven, charged or discharged as required by the OTG supplement. Writing a 1 to any of the bits 0 – 5 when they are active clears that bit and the corresponding interrupt. The USB_OTG_VBUS_IRQ register shares an interrupt source with USB_INTRUSB.

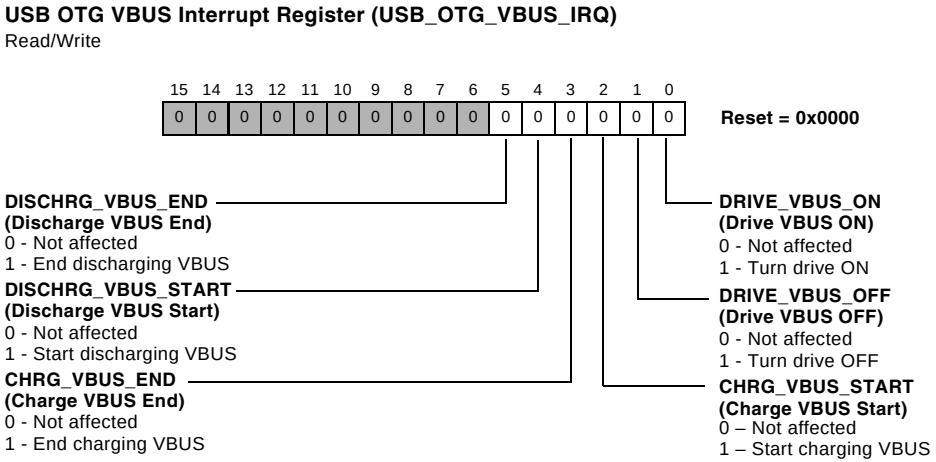

USB OTG VBUS Interrupt Register (USB_OTG_VBUS_IRQ)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DISCHRG_VBUS_END
(Discharge VBUS End)
0 - Not affected
1 - End discharging VBUS
DISCHRG_VBUS_START
(Discharge VBUS Start)
0 - Not affected
1 - Start discharging VBUS
CHRG_VBUS_END
(Charge VBUS End)
0 - Not affected
1 - End charging VBUS
DRIVE_VBUS_ON
(Drive VBUS ON)
0 - Not affected
1 - Turn drive ON
DRIVE_VBUS_OFF
(Drive VBUS OFF)
0 - Not affected
1 - Turn drive OFF
CHRG_VBUS_START
(Charge VBUS Start)
0 – Not affected
1 – Start charging VBUS

Figure 26-61. USB OTG VBUS Interrupt Register

Because the charge pump and VBUS charge/discharge circuit is located in a component/chip external to the on-chip PHY, the USB_OTG_VBUS_IRQ is provided as a means of allowing the software to drive the necessary control through a general-purpose, or dedicated IO.

DRIVE_VBUS_ON

When DRIVE_VBUS_ON (bit 0) is set, this status bit indicates the VBUS control circuit must be driven to greater than 4.4V ('A' device only).

DRIVE_VBUS_OFF

When DRIVE_VBUS_OFF (bit 1) is set, this status bit indicates the charge pump is to be shut off to end driving VBUS. ('A' device only).

CHRG_VBUS_START

When CHRG_VBUS_START (bit 2) is set, this status bit indicates the external control circuit is to begin charging VBUS to signal SRP ('B' device only).

CHRG_VBUS_END

When CHRG_VBUS_END (bit 3) is set, this status bit indicates the external VBUS control is to end charging of VBUS ('B' device only).

DISCHRG_VBUS_START

When DISCHRG_VBUS_START (bit 4) is set, this status bit indicates that VBUS is to be discharged in order to speed up VBUS discharging below SessionEnd threshold ('B' device only).

DISCHRG_VBUS_END

When DISCHRG_VBUS_END (bit 5) is set, this status bit indicates that VBUS control is to end discharging of VBUS ('B' device only).

USB OTG VBUS Mask (USB_OTG_VBUS_MASK) Register

The USB_OTG_VBUS_MASK register (see Figure 26-62) provides interrupt enable bits for the interrupt sources in USB_OTG_VBUS_IRQ.

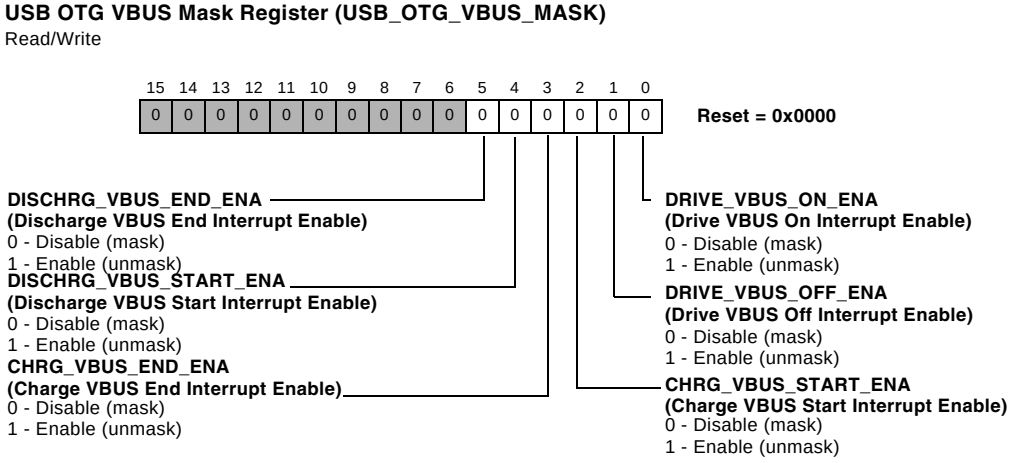

USB OTG VBUS Mask Register (USB_OTG_VBUS_MASK)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DISCHRG_VBUS_END_ENA
(Discharge VBUS End Interrupt Enable)
0 - Disable (mask)
1 - Enable (unmask)
DISCHRG_VBUS_START_ENA
(Discharge VBUS Start Interrupt Enable)
0 - Disable (mask)
1 - Enable (unmask)
CHRG_VBUS_END_ENA
(Charge VBUS End Interrupt Enable)
0 - Disable (mask)
1 - Enable (unmask)
DRIVE_VBUS_ON_ENA
(Drive VBUS On Interrupt Enable)
0 - Disable (mask)
1 - Enable (unmask)
DRIVE_VBUS_OFF_ENA
(Drive VBUS Off Interrupt Enable)
0 - Disable (mask)
1 - Enable (unmask)
CHRG_VBUS_START_ENA
(Charge VBUS Start Interrupt Enable)
0 - Disable (mask)
1 - Enable (unmask)

Figure 26-62. USB OTG VBUS Mask Register

USB Link Info (USB_LINKINFO) Register

The USB_LINKINFO register (see Figure 26-63) specifies PHY delays.

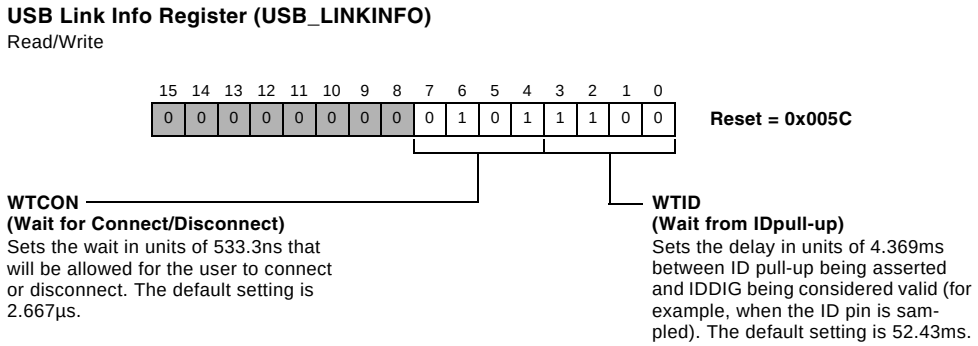

USB Link Info Register (USB_LINKINFO)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 0
Reset = 0x005C
WTCON
(Wait for Connect/Disconnect)
Sets the wait in units of 533.3ns that will be allowed for the user to connect or disconnect. The default setting is 2.667µs.
WTID
(Wait from IDpull-up)
Sets the delay in units of 4.369ms between ID pull-up being asserted and IDDIG being considered valid (for example, when the ID pin is sampled). The default setting is 52.43ms.

Figure 26-63. USB Link Info Register

USB VBUS Pulse Length (USB_VPLEN) Register

The USB_VPLEN register (see Figure 26-64) defines the duration of the VBUS pulsing charge for SRP initiation.

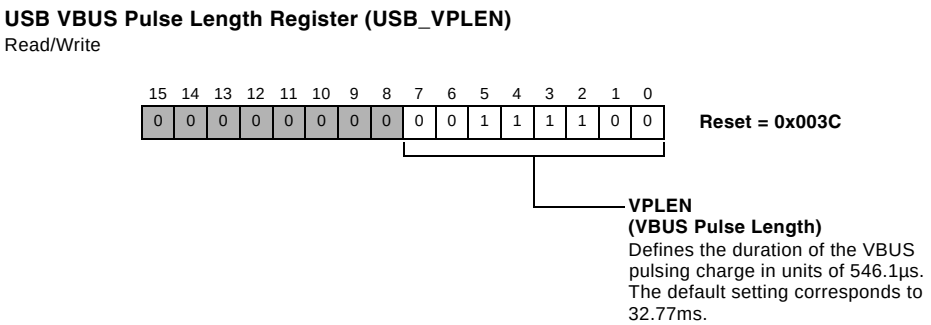

USB VBUS Pulse Length Register (USB_VPLEN)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0
Reset = 0x003C
VPLEN
(VBUS Pulse Length)
Defines the duration of the VBUS pulsing charge in units of 546.1µs. The default setting corresponds to 32.77ms.

Figure 26-64. USB VBUS Pulse Length Register

USB High-Speed EOF 1 (USB_HS_EOF1) Register

For high-speed transactions, the USB_HS_EOF1 register (see Figure 26-65) defines the minimum time gap allowed between the start of the last transaction and the EOF.

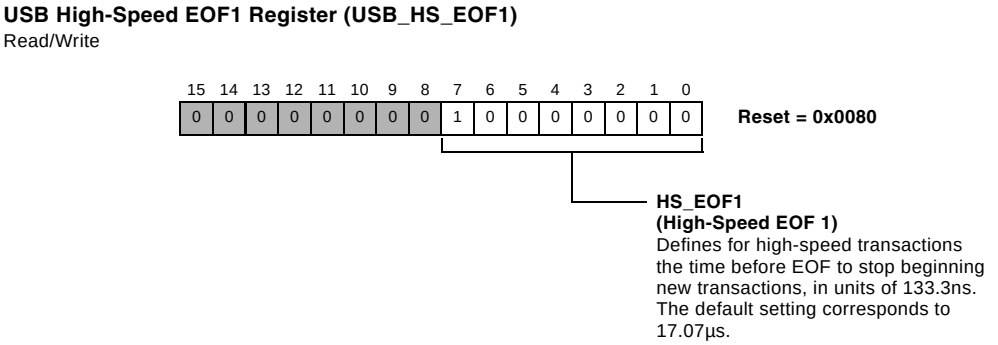

USB High-Speed EOF1 Register (USB_HS_EOF1)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
Reset = 0x0080
HS_EOF1
(High-Speed EOF 1)
Defines for high-speed transactions the time before EOF to stop beginning new transactions, in units of 133.3ns. The default setting corresponds to 17.07µs.

Figure 26-65. USB High-Speed EOF 1 Register

USB Full-Speed EOF 1 (USB_FS_EOF1) Register

For full-speed transactions, the USB_FS_EOF1 register (see Figure 26-66) defines the minimum time gap allowed between the start of the last transaction and the EOF.

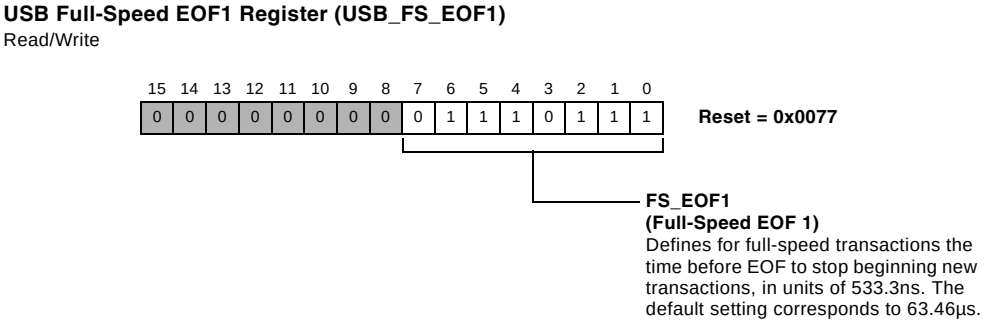

USB Full-Speed EOF1 Register (USB_FS_EOF1)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1
Reset = 0x0077
FS_EOF1
(Full-Speed EOF 1)
Defines for full-speed transactions the time before EOF to stop beginning new transactions, in units of 533.3ns. The default setting corresponds to 63.46µs.

Figure 26-66. USB Full-Speed EOF 1 Register

USB Low-Speed EOF 1 (USB_LS_EOF1) Register

For low-speed transactions, the `USB_LS_EOF1` register (see Figure 26-67) defines the minimum time gap allowed between the start of the last transaction and the EOF.

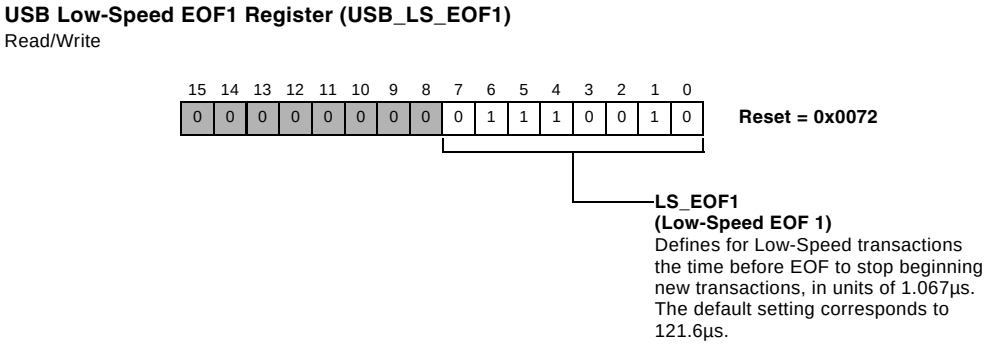

USB Low-Speed EOF1 Register (USB_LS_EOF1)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 0
Reset = 0x0072
LS_EOF1
(Low-Speed EOF 1)
Defines for Low-Speed transactions the time before EOF to stop beginning new transactions, in units of 1.067µs. The default setting corresponds to 121.6µs.

Figure 26-67. USB Low-Speed EOF 1 Register

USB APHY Control 2 (USB_APHY_CNTRL2) Register

The USB_APHY_CNTRL2 register (see Figure 26-68) controls transition of USB operation from normal to suspend to hibernate to resume normal.

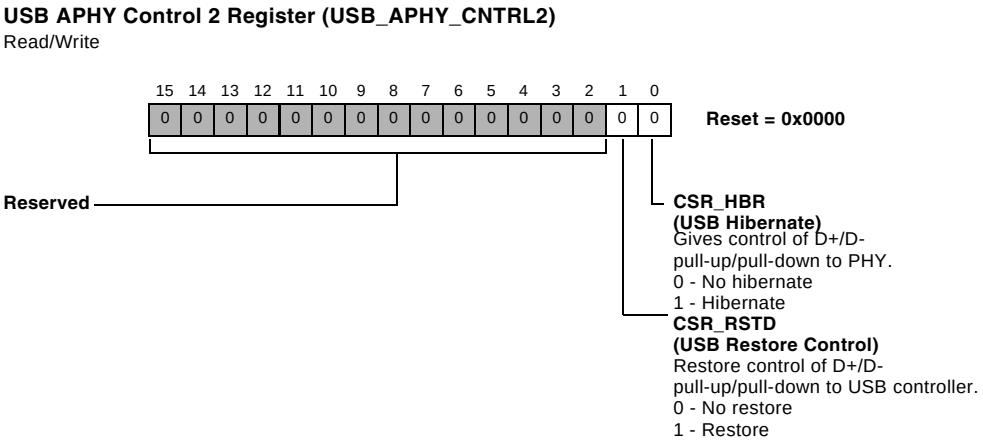

USB APHY Control 2 Register (USB_APHY_CNTRL2)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Reserved
CSR_HBR
(USB Hibernate)
Gives control of D+/D- pull-up/pull-down to PHY.
0 - No hibernate
1 - Hibernate
CSR_RSTD
(USB Restore Control)
Restore control of D+/D- pull-up/pull-down to USB controller.
0 - No restore
1 - Restore

Figure 26-68. USB APHY Control 2 Register

CSR_HBR

The CSR_HBR (bit 0) signals the analog PHY to hold the state of the pull-up and pull-downs on the D+ and D– for hibernate.

CSR_RSTD

The CSR_RSTD (bit 1) signals the analog PHY to release its hold on the D+/D– pull-ups and pull-downs and give control back to the USB controller.

USB PLL OSC Control (USB_PLLOSC_CTRL) Registers

The USB controller requires an internal clock of 960 MHz. This internal USB clock is generated from an external crystal. Recommended crystal values are 12 MHz, 24 MHz, and 30 MHz. Using a higher crystal fre-

quency will result in less jitter. The USB PLL multiplier select field (USB_MSEL) should be programmed so that the resulting USB PLL output frequency will be 960 MHz, as defined by the following equation:

$$\text{USB PLL output frequency} = (2 \times \text{USB\_MSEL} \times \text{CLKIN\_Freq})/(\text{DF} + 1)$$

where CLKIN_Freq is the value of the external crystal used. The reset value of this register clears the DF bit and sets the USB_MSEL to 20. Therefore, using a 24 MHz crystal will result in:

$$(2 \times 20 \times 24)/(0 + 1) = 960 \text{ MHz}$$

This register must be programmed before the USB module is enabled.

The USB_PLLOSC_CTRL register (see Figure 26-69) programs PLL and oscillator controls.

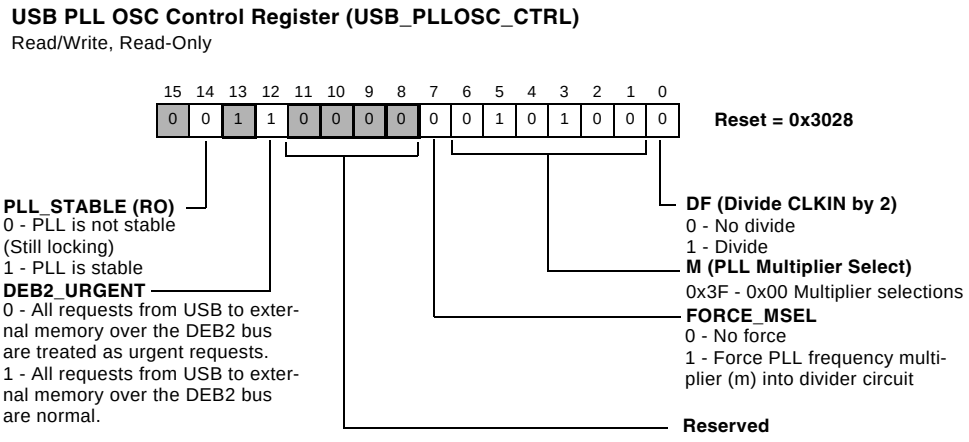

USB PLL OSC Control Register (USB_PLLOSC_CTRL)
Read/Write, Read-Only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 1 0 0 0 0 0 0 1 0 1 0 0 0
Reset = 0x3028
PLL_STABLE (RO)
0 - PLL is not stable
(Still locking)
1 - PLL is stable
DEB2_URGENT
0 - All requests from USB to external memory over the DEB2 bus are treated as urgent requests.
1 - All requests from USB to external memory over the DEB2 bus are normal.
DF (Divide CLKIN by 2)
0 - No divide
1 - Divide
M (PLL Multiplier Select)
0x3F - 0x00 Multiplier selections
FORCE_MSEL
0 - No force
1 - Force PLL frequency multiplier (m) into divider circuit
Reserved

Figure 26-69. USB PLL OSC Control Register

USB SRP Clock Divider (USB_SRP_CLKDIV) Register

The USB_SRP_CLKDIV register (see Figure 26-70) programs the clock divider for sleep recovery of the USB peripheral (wakeup from sleep mode).

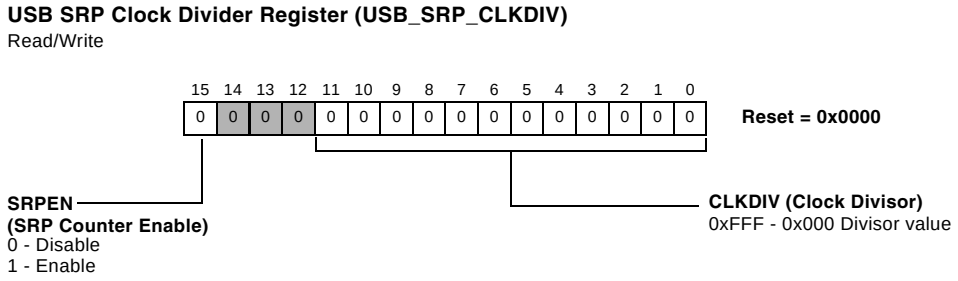

USB SRP Clock Divider Register (USB_SRP_CLKDIV)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SRPEN
(SRP Counter Enable)
0 - Disable
1 - Enable
CLKDIV (Clock Divisor)
0xFFF - 0x000 Divisor value

Figure 26-70. USB SRP Clock Divider Register

The processor is capable of running at peripheral clock frequencies up to 133 MHz. A 12-bit USB_SRP_CLKDIV register can be programmed to the desired value to divide the peripheral clock frequency that would clock the wakeup circuitry when the chip is put into sleep mode. For reliable operation of the circuit the user should program a value in the divider register that would divide the peripheral clock frequency greater than or equal to 32 kHz. The formula for calculating the value to be programmed into the USB_SRP_CLKDIV register is:

$$\frac{\text{SCLK frequency in kHz}}{32} - 1 = \text{CLKDIV value}$$

If SCLK = 130 MHz then CLKDIV = 4062 – 1 = 4061

If SCLK = 32 MHz then CLKDIV = 1000 – 1 = 999

USB DMA Interrupt (USB_DMA_INTERRUPT) Register

The USB_DMA_INTERRUPT register (see Figure 26-71) indicates which of the eight DMA master channels have a pending interrupt. The interrupt is generated when the corresponding DMA master channel DMA count register reaches zero. When the status is read by the processor core, software should write a 1 to the corresponding bit to clear the status.

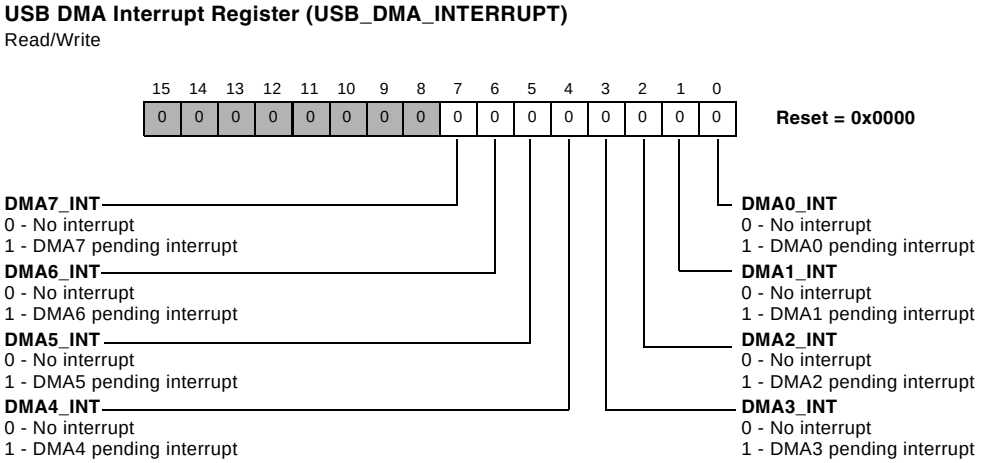

USB DMA Interrupt Register (USB_DMA_INTERRUPT)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DMA7_INT
0 - No interrupt
1 - DMA7 pending interrupt
DMA6_INT
0 - No interrupt
1 - DMA6 pending interrupt
DMA5_INT
0 - No interrupt
1 - DMA5 pending interrupt
DMA4_INT
0 - No interrupt
1 - DMA4 pending interrupt
DMA0_INT
0 - No interrupt
1 - DMA0 pending interrupt
DMA1_INT
0 - No interrupt
1 - DMA1 pending interrupt
DMA2_INT
0 - No interrupt
1 - DMA2 pending interrupt
DMA3_INT
0 - No interrupt
1 - DMA3 pending interrupt

Figure 26-71. USB DMA Interrupt Register

USB DMAx Control (USB_DMA_CONTROL) Registers

There is one USB_DMAx_CONTROL register (see Figure 26-72 on page 26-144) for each DMA master channel. DMA control is used to assign, configure and control each endpoint with a corresponding DMA

master channel. The n in the address below indicates the channel number 0 – 7.

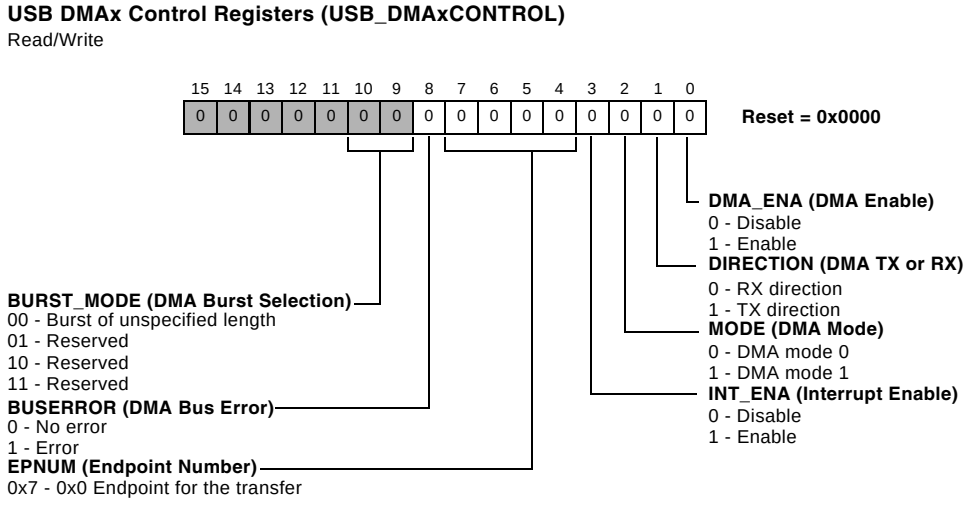

USB DMAx Control Registers (USB_DMAxCONTROL)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DMA_ENA (DMA Enable)
0 - Disable
1 - Enable
DIRECTION (DMA TX or RX)
0 - RX direction
1 - TX direction
MODE (DMA Mode)
0 - DMA mode 0
1 - DMA mode 1
INT_ENA (Interrupt Enable)
0 - Disable
1 - Enable
BURST_MODE (DMA Burst Selection)
00 - Burst of unspecified length
01 - Reserved
10 - Reserved
11 - Reserved
BUSERROR (DMA Bus Error)
0 - No error
1 - Error
EPNUM (Endpoint Number)
0x7 - 0x0 Endpoint for the transfer

Figure 26-72. USB DMAx Control Registers

DMA_ENA

DMA_ENA (bit 0) enables the corresponding DMA master channel to allow it to transfer data between the FIFOs and on-chip memory.

DIRECTION

DIRECTION (bit 1) determines the direction of the DMA transfer. A value of 0 indicates a DMA write (for use with RX endpoints), and a 1 indicates a DMA read (for use with TX endpoints).

MODE

MODE (bit 2) determines whether the channel operates in DMA mode 0 or DMA mode 1.

INT_ENA

INT_ENA (bit 3) enables DMA interrupts for that channel (enable bit for the corresponding bit in the USB_DMA_INTERRUPT register).

EPNUM

The EPNUM (bits 7–4) value indicates the endpoint that is to be used for the DMA transfer. The only values that are valid in this implementation are 0 through 7.

BUSERROR

BUSERROR (bit 8) indicates a peripheral bus error was encountered by the master channel.

BURST_MODE

BURST_MODE (bits 10–9) determine the type of burst transfer the corresponding DMA channel uses to transfer data.

USB DMAx Address Low (USB_DMAxADDRLOW) Registers

The USB_DMAxADDRLOW registers (see Figure 26-73) hold the least-significant half word of the full 32-bit DMA address. This indicates the location in on-chip memory where DMA data is written or read.

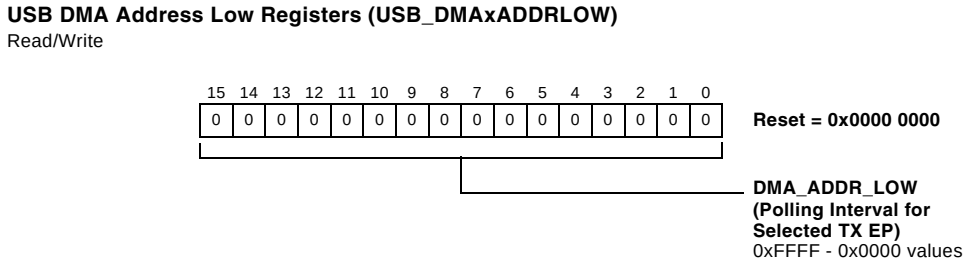

USB DMA Address Low Registers (USB_DMAxADDRLOW)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
DMA_ADDR_LOW
(Polling Interval for
Selected TX EP)
0xFFFF - 0x0000 values

Figure 26-73. USB DMAx Address Low Registers

USB DMAx Address High (USB_DMAxADDRHIGH) Registers

The USB_DMAxADDRHIGH registers (see Figure 26-74) hold the most-significant half word of the full 32-bit DMA address. This indicates the location in on-chip memory where DMA data is written or read.

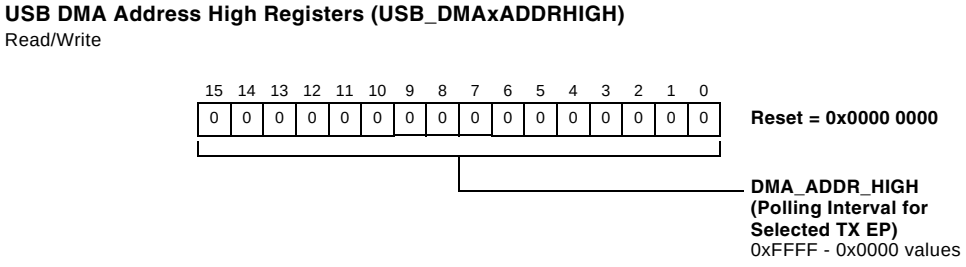

USB DMA Address High Registers (USB_DMAxADDRHIGH)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
DMA_ADDR_HIGH
(Polling Interval for
Selected TX EP)
0xFFFF - 0x0000 values

Figure 26-74. USB DMAx Address High Registers

USB DMAx Count Low (USB_DMAxCOUNTLOW) Registers

The USB_DMAxCOUNTLOW registers (see Figure 26-75) hold the least-significant half word of the full 32-bit DMA count for each DMA channel. The 32-bit DMA count indicates the number of bytes to be transferred for a given DMA work block.

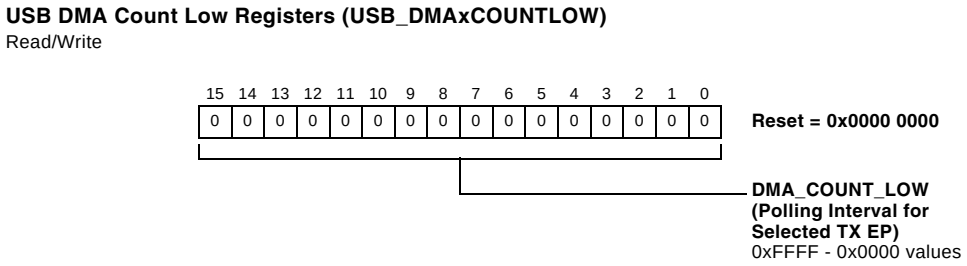

USB DMA Count Low Registers (USB_DMAxCOUNTLOW)
Read/Write
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
DMA_COUNT_LOW
(Polling Interval for
Selected TX EP)
0xFFFF - 0x0000 values

Figure 26-75. USB DMAx Count Low Registers

USB DMAx Count High (USB_DMAxCOUNTHIGH) Registers

The USB_DMAxCOUNTHIGH registers (see Figure 26-76 on page 26-148) hold the most-significant half word of the full 32-bit DMA count for each

DMA channel. The 32-bit DMA count indicates the number of bytes to be transferred for a given DMA work block.

USB DMA Count High Registers (USB_DMAxCOUNTHIGH)
Read/Write

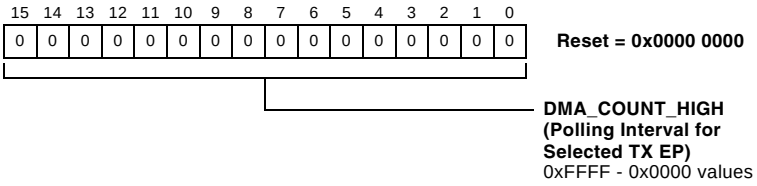

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
DMA_COUNT_HIGH
(Polling Interval for
Selected TX EP)
0xFFFF - 0x0000 values

Figure 26-76. USB DMAx Count High Registers

References

The following references provide further information regarding the USB.

- *On-The-Go Supplement to the USB 2.0 Specification*, Rev 1.0a, June 24, 2003, USB-IF
- *Universal Serial Bus Specification 2.0*

Glossary of USB Terms

A list of common USB terms and their definitions as used in this specification and with respect to the USB controller follows:

'A' Device

The USB device with a mini-A plug inserted into its receptacle. The 'A' device always supplies power to VBUS.

'B' Device

The USB device with a standard-B or mini-B plug inserted into its receptacle. The 'B' device starts a session as the peripheral.

Bi-directional endpoint

An endpoint that can concurrently support receive and transfer packets.

Control endpoint

An endpoint that is solely used for transfer of USB control packets for setup and configuration. In all USB devices, the control endpoint refers to the bi-directional endpoint 0.

Dual role device

A USB device that can operate either as the USB host in an OTG session or as a traditional USB peripheral.

Endpoint

A single physical communication channel for USB, implemented as a FIFO and control logic for that endpoint. Each endpoint has an associated USB transfer type, maximum packet size, bandwidth requirement, endpoint number, and (often) a fixed transfer direction.

Frame

A regular, fixed 1ms time slot that can contain several transactions. The transfer type determines what transactions are permitted for a given endpoint.

HNP

Host negotiation protocol. Part of the USB OTG Supplement that allows the host function to be transferred between two connected dual role devices.

Packet

The lowest level of data exchange on USB. The size is determined by the transfer type and buffer size of the USB peripheral.

PHY

The PHY is a transceiver circuit that implements the physical layer of USB. For full speed USB OTG this includes line drivers and receivers, pull-up/pull-down resistors as well as device ID and VBUS level detection.

Session

A period during which USB transfers take place within an OTG connection. This can be initiated by the 'A' device (by driving VBUS) or 'B' device (by initiating SRP). VBUS is powered during a session.

SRP

Session request protocol. Part of the USB OTG Supplement that allows a 'B' device to turn on VBUS and initiate a USB session.

Transaction

Collection of one or more packets in sequence

Transfer

Collection of one or more transfers in sequence

Uni-directional endpoint

Endpoint with its direction fixed in a single direction (for example, it can only receive packets from the USB) in both host and peripheral modes.

A SYSTEM MMR ASSIGNMENTS

This appendix lists MMR addresses and register names for the system memory-mapped registers (MMRs), the core timer registers, and the processor-specific memory registers mentioned in this manual.

This appendix contains:

- "Processor-Specific Memory Registers" on page A-35
- "OTP Memory Registers" on page A-5
- "System Reset and Interrupt Control Registers" on page A-4
- "DMA/Memory DMA Control Registers" on page A-15
- "Handshake MDMA Control Registers" on page A-24
- "External Bus Interface Unit Registers" on page A-14
- "HOST DMA Port Registers" on page A-25
- "Ports Registers" on page A-9
- "Timer Registers" on page A-8
- "Core Timer Registers" on page A-35
- "Watchdog Timer Registers" on page A-5
- "GP Counter Registers" on page A-25
- "Real-Time Clock Registers" on page A-6
- "PPI Registers" on page A-17

- "Security Registers" on page A-18
- "Reset and Booting Registers" on page A-18
- "Dynamic Power Management Registers" on page A-3
- "NFC Registers" on page A-26
- "Ethernet MAC Registers" on page A-20
- "SPI Controller Registers" on page A-7
- "TWI Registers" on page A-19
- "SPORT0 Controller Registers" on page A-12
- "SPORT1 Controller Registers" on page A-13
- "UART0 Controller Registers" on page A-6
- "UART1 Controller Registers" on page A-20
- "USB Registers" on page A-27

Registers are listed in order by their memory-mapped address. To find more information about an MMR, refer to the page shown in the "Description" column.

These notes provide general information about the system memory-mapped registers (MMRs):

- The system MMR address range is 0xFFC0 0000 – 0xFFDF FFFF.
- All system MMRs are either 16 bits or 32 bits wide. MMRs that are 16 bits wide must be accessed with 16-bit read or write operations. MMRs that are 32 bits wide must be accessed with 32-bit read or write operations. Check the description of the MMR to determine whether a 16-bit or a 32-bit access is required.
- All system MMR space that is not defined in this appendix is reserved for internal use only.

Dynamic Power Management Registers

Dynamic power management registers (0xFFC0 0000 – 0xFFC0 00FF) are listed in Table A-1.

Table A-1. Dynamic Power Management Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0000	PLL_CTL	"PLL Control Register for ADSP-BF523/5/7" on page 18-26
0xFFC0 0004	PLL_DIV	"PLL Divide Register" on page 18-26
0xFFC0 0008	VR_CTL	"ADSP-BF523/525/527 Voltage Regulator Control Register" on page 18-29
0xFFC0 000C	PLL_STAT	"PLL Status Register" on page 18-28
0xFFC0 0010	PLL_LOCKCNT	"PLL Lock Count Register" on page 18-28

System Reset and Interrupt Control Registers

System reset and interrupt control registers (0xFFC0 0100 – 0xFFC0 01FF) are listed in Table A-2.

Table A-2. System Reset and Interrupt Control Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0104	SYSCR	"System Reset Configuration (SYSCR) Register" on page 17-100
0xFFC0 010C	SIC_IMASK0	"System Interrupt Mask (SIC_IMASK) Register" on page 5-12
0xFFC0 014C	SIC_IMASK1	"System Interrupt Mask (SIC_IMASK) Register" on page 5-12
0xFFC0 0110	SIC_IAR0	"System Interrupt Assignment Register" on page 5-11
0xFFC0 0114	SIC_IAR1	"System Interrupt Assignment Register" on page 5-11
0xFFC0 0118	SIC_IAR2	"System Interrupt Assignment Register" on page 5-11
0xFFC0 011C	SIC_IAR3	"System Interrupt Assignment Register" on page 5-11
0xFFC0 0150	SIC_IAR4	"System Interrupt Assignment Register" on page 5-11
0xFFC0 0154	SIC_IAR5	"System Interrupt Assignment Register" on page 5-11
0xFFC0 0158	SIC_IAR6	"System Interrupt Assignment Register" on page 5-11
0xFFC0 015C	SIC_IAR7	"System Interrupt Assignment Register" on page 5-11
0xFFC0 0120	SIC_ISR0	"System Interrupt Status (SIC_ISR) Register" on page 5-12
0xFFC0 0160	SIC_ISR1	"System Interrupt Status (SIC_ISR) Register" on page 5-12
0xFFC0 0124	SIC_IWR0	"System Interrupt Wakeup-Enable (SIC_IWR) Register" on page 5-12
0xFFC0 0164	SIC_IWR1	"System Interrupt Wakeup-Enable (SIC_IWR) Register" on page 5-12

OTP Memory Registers

Table A-3. Watchdog Timer Registers

Memory Mapped Address	Register Name	Description
	OTP_TIMING	"OTP_TIMING Register" on page 4-17
		"Error Codes" on page 4-24

Watchdog Timer Registers

Watchdog timer registers (0xFFC0 0200 – 0xFFC0 02FF) are listed in Table A-4.

Table A-4. Watchdog Timer Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0200	WDOG_CTL	"Watchdog Control (WDOG_CTL) Register" on page 12-7
0xFFC0 0204	WDOG_CNT	"Watchdog Count (WDOG_CNT) Register" on page 12-5
0xFFC0 0208	WDOG_STAT	"Watchdog Status (WDOG_STAT) Register" on page 12-6

Real-Time Clock Registers

Real-time clock registers (0xFFC0 0300 – 0xFFC0 03FF) are listed in Table A-5.

Table A-5. Real-Time Clock Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0300	RTC_STAT	"RTC Status Register" on page 14-20
0xFFC0 0304	RTC_ICTL	"RTC Interrupt Control Register" on page 14-20
0xFFC0 0308	RTC_ISTAT	"RTC Interrupt Status Register" on page 14-21
0xFFC0 030C	RTC_SWCNT	"RTC Stopwatch Count Register" on page 14-21
0xFFC0 0310	RTC_ALARM	"RTC Alarm Register" on page 14-22
0xFFC0 0314	RTC_PREN	"RTC Prescaler Enable Register" on page 14-22

UART0 Controller Registers

UART0 controller registers (0xFFC0 0400 – 0xFFC0 04FF) are listed in Table A-6. For UART1 registers, see Table A-20 on page A-20.

Table A-6. UART0 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0400	UART0_THR	"UART Transmit Holding Register" on page 25-27
0xFFC0 0400	UART0_RBR	"UART Receive Buffer Register" on page 25-27
0xFFC0 0400	UART0_DLL	"UART Divisor Latch Registers" on page 25-31
0xFFC0 0404	UART0_DLH	"UART Divisor Latch Registers" on page 25-31
0xFFC0 0404	UART0_IER	"UART Interrupt Enable Register" on page 25-28
0xFFC0 0408	UART0_IIR	"UART Interrupt Identification Register" on page 25-30
0xFFC0 040C	UART0_LCR	"UART Line Control Register" on page 25-22

Table A-6. UART0 Controller Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 0410	UART0_MCR	"UART Modem Control Registers" on page 25-24
0xFFC0 0414	UART0_LSR	"UART Line Status Register" on page 25-25
0xFFC0 041C	UART0_SCR	"UART Scratch Register" on page 25-32
0xFFC0 0424	UART0_GCTL	"UART Global Control Register" on page 25-32

SPI Controller Registers

SPI controller registers (0xFFC0 0500 – 0xFFC0 05FF) are listed in Table A-7.

Table A-7. SPI Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0500	SPI_CTL	"SPI Control Register" on page 22-37
0xFFC0 0504	SPI_FLG	"SPI Flag Register (example with 7 slave selects)" on page 22-38
0xFFC0 0508	SPI_STAT	"SPI Status Register" on page 22-40
0xFFC0 050C	SPI_TDBR	"SPI Transmit Data Buffer Register" on page 22-43
0xFFC0 0510	SPI_RDBR	"SPI Receive Data Buffer Register" on page 22-44
0xFFC0 0514	SPI_BAUD	"SPI Baud Rate Register" on page 22-35
0xFFC0 0518	SPI_SHADOW	"SPI RDBR Shadow Register" on page 22-44

Timer Registers

Timer registers (0xFFC0 0600 – 0xFFC0 06FF) are listed in Table A-8.

Table A-8. Timer Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0600	TIMER0_CONFIG	"Timer Configuration Register" on page 10-41
0xFFC0 0604	TIMER0_COUNTER	"Timer Counter Register" on page 10-43
0xFFC0 0608	TIMER0_PERIOD	"Timer Period Register" on page 10-45
0xFFC0 060C	TIMER0_WIDTH	"Timer Width Register" on page 10-45
0xFFC0 0610	TIMER1_CONFIG	"Timer Configuration Register" on page 10-41
0xFFC0 0614	TIMER1_COUNTER	"Timer Counter Register" on page 10-43
0xFFC0 0618	TIMER1_PERIOD	"Timer Period Register" on page 10-45
0xFFC0 061C	TIMER1_WIDTH	"Timer Width Register" on page 10-45
0xFFC0 0620	TIMER2_CONFIG	"Timer Configuration Register" on page 10-41
0xFFC0 0624	TIMER2_COUNTER	"Timer Counter Register" on page 10-43
0xFFC0 0628	TIMER2_PERIOD	"Timer Period Register" on page 10-45
0xFFC0 062C	TIMER2_WIDTH	"Timer Width Register" on page 10-45
0xFFC0 0630	TIMER3_CONFIG	"Timer Configuration Register" on page 10-41
0xFFC0 0634	TIMER3_COUNTER	"Timer Counter Register" on page 10-43
0xFFC0 0638	TIMER3_PERIOD	"Timer Period Register" on page 10-45
0xFFC0 063C	TIMER3_WIDTH	"Timer Width Register" on page 10-45
0xFFC0 0640	TIMER4_CONFIG	"Timer Configuration Register" on page 10-41
0xFFC0 0644	TIMER4_COUNTER	"Timer Counter Register" on page 10-43
0xFFC0 0648	TIMER4_PERIOD	"Timer Period Register" on page 10-45
0xFFC0 064C	TIMER4_WIDTH	"Timer Width Register" on page 10-45
0xFFC0 0650	TIMER5_CONFIG	"Timer Configuration Register" on page 10-41
0xFFC0 0654	TIMER5_COUNTER	"Timer Counter Register" on page 10-43
0xFFC0 0658	TIMER5_PERIOD	"Timer Period Register" on page 10-45
0xFFC0 065C	TIMER5_WIDTH	"Timer Width Register" on page 10-45

Table A-8. Timer Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 0660	TIMER6_CONFIG	"Timer Configuration Register" on page 10-41
0xFFC0 0664	TIMER6_COUNTER	"Timer Counter Register" on page 10-43
0xFFC0 0668	TIMER6_PERIOD	"Timer Period Register" on page 10-45
0xFFC0 066C	TIMER6_WIDTH	"Timer Width Register" on page 10-45
0xFFC0 0670	TIMER7_CONFIG	"Timer Configuration Register" on page 10-41
0xFFC0 0674	TIMER7_COUNTER	"Timer Counter Register" on page 10-43
0xFFC0 0678	TIMER7_PERIOD	"Timer Period Register" on page 10-45
0xFFC0 067C	TIMER7_WIDTH	"Timer Width Register" on page 10-45
0xFFC0 0680	TIMER_ENABLE	"Timer Enable Register" on page 10-36
0xFFC0 0684	TIMER_DISABLE	"Timer Disable Register" on page 10-37
0xFFC0 0688	TIMER_STATUS	"Timer Status Register" on page 10-39

Ports Registers

Ports registers (port F: 0xFFC0 0700 – 0xFFC0 07FF, port G: 0xFFC0 1500 – 0xFFC0 15FF, port H: 0xFFC0 1700 – 0xFFC0 17FF, pin control: 0xFFC0 3200 – 0xFFC0 32FF) are listed in Table A-9.

Table A-9. Ports Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0700	PORTFIO	"GPIO Data Registers" on page 9-32
0xFFC0 0704	PORTFIO_CLEAR	"GPIO Clear Registers" on page 9-33
0xFFC0 0708	PORTFIO_SET	"GPIO Set Registers" on page 9-33
0xFFC0 070C	PORTFIO_TOGGLE	"GPIO Toggle Registers" on page 9-34
0xFFC0 0710	PORTFIO_MASKA	"GPIO Mask Interrupt A Registers" on page 9-36
0xFFC0 0714	PORTFIO_MASKA_CLEAR	"GPIO Mask Interrupt A Clear Registers" on page 9-39

Table A-9. Ports Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 0718	PORTFIO_MASKA_SET	"GPIO Mask Interrupt A Set Registers" on page 9-37
0xFFC0 071C	PORTFIO_MASKA_TOGGLE	"GPIO Mask Interrupt A Toggle Registers" on page 9-41
0xFFC0 0720	PORTFIO_MASKB	"GPIO Mask Interrupt B Registers" on page 9-36
0xFFC0 0724	PORTFIO_MASKB_CLEAR	"GPIO Mask Interrupt B Clear Registers" on page 9-40
0xFFC0 0728	PORTFIO_MASKB_SET	"GPIO Mask Interrupt B Set Registers" on page 9-38
0xFFC0 072C	PORTFIO_MASKB_TOGGLE	"GPIO Mask Interrupt B Toggle Registers" on page 9-42
0xFFC0 0730	PORTFIO_DIR	"GPIO Direction Registers" on page 9-31
0xFFC0 0734	PORTFIO_POLAR	"GPIO Polarity Registers" on page 9-34
0xFFC0 0738	PORTFIO_EDGE	"Interrupt Sensitivity Registers" on page 9-35
0xFFC0 073C	PORTFIO_BOTH	"GPIO Set on Both Edges Registers" on page 9-35
0xFFC0 0740	PORTFIO_INEN	"GPIO Input Enable Registers" on page 9-32
0xFFC0 1500	PORTGIO	"GPIO Data Registers" on page 9-32
0xFFC0 1504	PORTGIO_CLEAR	"GPIO Clear Registers" on page 9-33
0xFFC0 1508	PORTGIO_SET	"GPIO Set Registers" on page 9-33
0xFFC0 150C	PORTGIO_TOGGLE	"GPIO Toggle Registers" on page 9-34
0xFFC0 1510	PORTGIO_MASKA	"GPIO Mask Interrupt A Registers" on page 9-36
0xFFC0 1514	PORTGIO_MASKA_CLEAR	"GPIO Mask Interrupt A Clear Registers" on page 9-39
0xFFC0 1518	PORTGIO_MASKA_SET	"GPIO Mask Interrupt A Set Registers" on page 9-37
0xFFC0 151C	PORTGIO_MASKA_TOGGLE	"GPIO Mask Interrupt A Toggle Registers" on page 9-41
0xFFC0 1520	PORTGIO_MASKB	"GPIO Mask Interrupt B Registers" on page 9-36
0xFFC0 1524	PORTGIO_MASKB_CLEAR	"GPIO Mask Interrupt B Clear Registers" on page 9-40
0xFFC0 1528	PORTGIO_MASKB_SET	"GPIO Mask Interrupt B Set Registers" on page 9-38
0xFFC0 152C	PORTGIO_MASKB_TOGGLE	"GPIO Mask Interrupt B Toggle Registers" on page 9-42
0xFFC0 1530	PORTGIO_DIR	"GPIO Direction Registers" on page 9-31
0xFFC0 1534	PORTGIO_POLAR	"GPIO Polarity Registers" on page 9-34
0xFFC0 1538	PORTGIO_EDGE	"Interrupt Sensitivity Registers" on page 9-35
0xFFC0 153C	PORTGIO_BOTH	"GPIO Set on Both Edges Registers" on page 9-35
0xFFC0 1540	PORTGIO_INEN	"GPIO Input Enable Registers" on page 9-32

Table A-9. Ports Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 1700	PORTHIO	"GPIO Data Registers" on page 9-32
0xFFC0 1704	PORTHIO_CLEAR	"GPIO Clear Registers" on page 9-33
0xFFC0 1708	PORTHIO_SET	"GPIO Set Registers" on page 9-33
0xFFC0 170C	PORTHIO_TOGGLE	"GPIO Toggle Registers" on page 9-34
0xFFC0 1710	PORTHIO_MASKA	"GPIO Mask Interrupt A Registers" on page 9-36
0xFFC0 1714	PORTHIO_MASKA_CLEAR	"GPIO Mask Interrupt A Clear Registers" on page 9-39
0xFFC0 1718	PORTHIO_MASKA_SET	"GPIO Mask Interrupt A Set Registers" on page 9-37
0xFFC0 171C	PORTHIO_MASKA_TOGGLE	"GPIO Mask Interrupt A Toggle Registers" on page 9-41
0xFFC0 1720	PORTHIO_MASKB	"GPIO Mask Interrupt B Registers" on page 9-36
0xFFC0 1724	PORTHIO_MASKB_CLEAR	"GPIO Mask Interrupt B Clear Registers" on page 9-40
0xFFC0 1728	PORTHIO_MASKB_SET	"GPIO Mask Interrupt B Set Registers" on page 9-38
0xFFC0 172C	PORTHIO_MASKB_TOGGLE	"GPIO Mask Interrupt B Toggle Registers" on page 9-42
0xFFC0 1730	PORTHIO_DIR	"GPIO Direction Registers" on page 9-31
0xFFC0 1734	PORTHIO_POLAR	"GPIO Polarity Registers" on page 9-34
0xFFC0 1738	PORTHIO_EDGE	"Interrupt Sensitivity Registers" on page 9-35
0xFFC0 173C	PORTHIO_BOTH	"GPIO Set on Both Edges Registers" on page 9-35
0xFFC0 1740	PORTHIO_INEN	"GPIO Input Enable Registers" on page 9-32
0xFFC0 3200	PORTF_FER	"Function Enable Registers" on page 9-30
0xFFC0 3204	PORTG_FER	"Function Enable Registers" on page 9-30
0xFFC0 3208	PORTH_FER	"Function Enable Registers" on page 9-30
0xFFC0 3210	PORTF_MUX	"Port F Multiplexer Control Register" on page 9-28
0xFFC0 3214	PORTG_MUX	"Port F Multiplexer Control Register" on page 9-28
0xFFC0 3218	PORTH_MUX	"Port F Multiplexer Control Register" on page 9-28
0xFFC0 3240	PORTF_HYSTERESIS	"Port F Hysteresis Register" on page 9-24
0xFFC0 3244	PORTG_HYSTERESIS	"Port G Hysteresis Register" on page 9-25
0xFFC0 3248	PORTH_HYSTERESIS	"Port H Hysteresis Register" on page 9-25
0xFFC0 3288	NONGPIO_HYSTERESIS	"Non-GPIO Hysteresis Control Register" on page 9-26
	NONGPIO_DRIVE	"TWI Drive Strength Control Register" on page 9-27

SPORT0 Controller Registers

SPORT0 controller registers (0xFFC0 0800 – 0xFFC0 08FF) are listed in Table A-10.

Table A-10. SPORT0 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0800	SPORT0_TCR1	"SPORT Transmit Configuration 1 Register" on page 24-49
0xFFC0 0804	SPORT0_TCR2	"SPORT Transmit Configuration 2 Register" on page 24-50
0xFFC0 0808	SPORT0_TCLKDIV	"SPORT Transmit Serial Clock Divider Register" on page 24-65
0xFFC0 080C	SPORT0_TFSDIV	"SPORT Transmit Frame Sync Divider Register" on page 24-66
0xFFC0 0810	SPORT0_TX	"SPORT Transmit Data Register" on page 24-61
0xFFC0 0818	SPORT0_RX	"SPORT Receive Data Register" on page 24-63
0xFFC0 0820	SPORT0_RCR1	"SPORT Receive Configuration 1 Register" on page 24-55
0xFFC0 0824	SPORT0_RCR2	"SPORT Receive Configuration 2 Register" on page 24-56
0xFFC0 0828	SPORT0_RCLKDIV	"SPORT Receive Serial Clock Divider Register" on page 24-66
0xFFC0 082C	SPORT0_RFSDIV	"SPORT Receive Frame Sync Divider Register" on page 24-67
0xFFC0 0830	SPORT0_STAT	"SPORT Status Register" on page 24-65
0xFFC0 0834	SPORT0_CHNL	"SPORT Current Channel Register" on page 24-69
0xFFC0 0838	SPORT0_MCMC1	"SPORT Multichannel Configuration Register 1" on page 24-67
0xFFC0 083C	SPORT0_MCMC2	"SPORT Multichannel Configuration Register 2" on page 24-68
0xFFC0 0840	SPORT0_MTCS0	"SPORT Multichannel Transmit Select Registers" on page 24-71
0xFFC0 0844	SPORT0_MTCS1	"SPORT Multichannel Transmit Select Registers" on page 24-71
0xFFC0 0848	SPORT0_MTCS2	"SPORT Multichannel Transmit Select Registers" on page 24-71
0xFFC0 084C	SPORT0_MTCS3	"SPORT Multichannel Transmit Select Registers" on page 24-71
0xFFC0 0850	SPORT0_MRCS0	"SPORT Multichannel Receive Select Registers" on page 24-70
0xFFC0 0854	SPORT0_MRCS1	"SPORT Multichannel Receive Select Registers" on page 24-70
0xFFC0 0858	SPORT0_MRCS2	"SPORT Multichannel Receive Select Registers" on page 24-70
0xFFC0 085C	SPORT0_MRCS3	"SPORT Multichannel Receive Select Registers" on page 24-70

SPORT1 Controller Registers

SPORT1 controller registers (0xFFC0 0900 – 0xFFC0 09FF) are listed in Table A-11.

Table A-11. SPORT 1 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0900	SPORT1_TCR1	"SPORT Transmit Configuration 1 Register" on page 24-49
0xFFC0 0904	SPORT1_TCR2	"SPORT Transmit Configuration 2 Register" on page 24-50
0xFFC0 0908	SPORT1_TCLKDIV	"SPORT Transmit Serial Clock Divider Register" on page 24-65
0xFFC0 090C	SPORT1_TFSDIV	"SPORT Transmit Frame Sync Divider Register" on page 24-66
0xFFC0 0910	SPORT1_TX	"SPORT Transmit Data Register" on page 24-61
0xFFC0 0918	SPORT1_RX	"SPORT Receive Data Register" on page 24-63
0xFFC0 0920	SPORT1_RCR1	"SPORT Receive Configuration 1 Register" on page 24-55
0xFFC0 0924	SPORT1_RCR2	"SPORT Receive Configuration 2 Register" on page 24-56
0xFFC0 0928	SPORT1_RCLKDIV	"SPORT Receive Serial Clock Divider Register" on page 24-66
0xFFC0 092C	SPORT1_RFSDIV	"SPORT Receive Frame Sync Divider Register" on page 24-67
0xFFC0 0930	SPORT1_STAT	"SPORT Status Register" on page 24-65
0xFFC0 0934	SPORT1_CHNL	"SPORT Current Channel Register" on page 24-69
0xFFC0 0938	SPORT1_MCMC1	"SPORT Multichannel Configuration Register 1" on page 24-67
0xFFC0 093C	SPORT1_MCMC2	"SPORT Multichannel Configuration Register 2" on page 24-68
0xFFC0 0940	SPORT1_MTCS0	"SPORT Multichannel Transmit Select Registers" on page 24-71
0xFFC0 0944	SPORT1_MTCS1	"SPORT Multichannel Transmit Select Registers" on page 24-71
0xFFC0 0948	SPORT1_MTCS2	"SPORT Multichannel Transmit Select Registers" on page 24-71
0xFFC0 094C	SPORT1_MTCS3	"SPORT Multichannel Transmit Select Registers" on page 24-71
0xFFC0 0950	SPORT1_MRCS0	"SPORT Multichannel Receive Select Registers" on page 24-70
0xFFC0 0954	SPORT1_MRCS1	"SPORT Multichannel Receive Select Registers" on page 24-70
0xFFC0 0958	SPORT1_MRCS2	"SPORT Multichannel Receive Select Registers" on page 24-70
0xFFC0 095C	SPORT1_MRCS3	"SPORT Multichannel Receive Select Registers" on page 24-70

External Bus Interface Unit Registers

External bus interface unit registers (0xFFC0 0A00 – 0xFFC0 0AFF) are listed in Table A-12.

Table A-12. External Bus Interface Unit Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0A00	EBIU_AMGCTL	"Asynchronous Memory Global Control Register" on page 7-20
0xFFC0 0A04	EBIU_AMBCTL0	"Asynchronous Memory Bank Control 0 Register" on page 7-21
0xFFC0 0A08	EBIU_AMBCTL1	"Asynchronous Memory Bank Control 1 Register" on page 7-22
0xFFC0 0A10	EBIU_SDGCTL	"SDRAM Memory Global Control Register" on page 7-66
0xFFC0 0A14	EBIU_SDBCTL	"SDRAM Memory Bank Control Register" on page 7-62
0xFFC0 0A18	EBIU_SDRRC	"SDRAM Refresh Rate Control Register" on page 7-59
0xFFC0 0A1C	EBIU_SDSTAT	"SDRAM Control Status Register" on page 7-75

DMA/Memory DMA Control Registers

DMA control registers (0xFFC0 0B00 – 0xFFC0 0FFF) are listed in Table A-13.

Table A-13. DMA Traffic Control Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0B0C	DMA_TC_PER	"DMA Traffic Control Counter Period Register" on page 6-91
0xFFC0 0B10	DMA_TC_CNT	"DMA Traffic Control Counter Register" on page 6-92

Since each DMA channel has an identical MMR set, with fixed offsets from the base address associated with that DMA channel, it is convenient to view the MMR information as provided in Table A-14 and Table A-15.

Table A-14 identifies the base address of each DMA channel, as well as the register prefix that identifies the channel. Table A-15 then lists the register suffix and provides its offset from the Base Address.

As an example, the DMA channel 0 Y_MODIFY register is called `DMA0_Y_MODIFY`, and its address is 0xFFC0 0C1C. Likewise, the memory DMA stream 0 source current address register is called `MDMA_S0_CURR_ADDR`, and its address is 0xFFC0 0E64.

Table A-14. DMA Channel Base Addresses

DMA Channel Identifier	MMR Base Address	Register Prefix
0	0xFFC0 0C00	DMA0_
1	0xFFC0 0C40	DMA1_
2	0xFFC0 0C80	DMA2_
3	0xFFC0 0CC0	DMA3_
4	0xFFC0 0D00	DMA4_
5	0xFFC0 0D40	DMA5_
6	0xFFC0 0D80	DMA6_
7	0xFFC0 0DC0	DMA7_
8	0xFFC0 0E00	DMA8_
9	0xFFC0 0E40	DMA9_
10	0xFFC0 0E80	DMA10_
11	0xFFC0 0EC0	DMA11_
MemDMA stream 0 destination	0xFFC0 0F00	MDMA_D0_
MemDMA stream 0 source	0xFFC0 0F40	MDMA_S0_
MemDMA stream 1 destination	0xFFC0 0F80	MDMA_D1_
MemDMA stream 1 source	0xFFC0 0FC0	MDMA_S1_

Table A-15. DMA Register Suffix and Offset

Register Suffix	Offset From Base	Description
NEXT_DESC_PTR	0x00	"DMA Next Descriptor Pointer Registers" on page 6-83
START_ADDR	0x04	"DMA Start Address Registers" on page 6-76

Table A-15. DMA Register Suffix and Offset (Continued)

Register Suffix	Offset From Base	Description
CONFIG	0x08	"DMA Configuration Registers" on page 6-69
X_COUNT	0x10	"DMA Inner Loop Count Registers" on page 6-78
X_MODIFY	0x14	"DMA Inner Loop Address Increment Registers" on page 6-80
Y_COUNT	0x18	"DMA Outer Loop Count Registers" on page 6-80
Y_MODIFY	0x1C	"DMA Outer Loop Address Increment Registers" on page 6-82
CURR_DESC_PTR	0x20	"DMA Current Descriptor Pointer Registers" on page 6-84
CURR_ADDR	0x24	"DMA Current Address Registers" on page 6-77
IRQ_STATUS	0x28	"DMA Interrupt Status Registers" on page 6-74
PERIPHERAL_MAP	0x2C	"DMA Peripheral Map Registers" on page 6-68
CURR_X_COUNT	0x30	"DMA Current Inner Loop Count Registers" on page 6-79
CURR_Y_COUNT	0x38	"DMA Current Outer Loop Count Registers" on page 6-81

PPI Registers

PPI registers (0xFFC0 1000 – 0xFFC0 10FF) are listed in Table A-16.

Table A-16. PPI Registers

Memory Mapped Address	Register Name	Description
0xFFC0 1000	PPI_CONTROL	"PPI Control Register" on page 15-27
0xFFC0 1004	PPI_STATUS	"PPI Status Register" on page 15-32
0xFFC0 1008	PPI_COUNT	"PPI Transfer Count Register" on page 15-35
0xFFC0 100C	PPI_DELAY	"PPI Delay Count Register" on page 15-34
0xFFC0 1010	PPI_FRAME	"PPI Lines Per Frame Register" on page 15-36

Security Registers

Table A-17. Security Registers

Memory Mapped Address	Register Name	Description
0xFFC03620	SECURE_SYSSWT	"Secure System Switch (SECURE_SYSSWT) Register" on page 16-46
0xFFC03624	SECURE_CONTROL	"Secure Control (SECURE_CONTROL) Register" on page 16-52
0xFFC03628	SECURE_STATUS	"Secure Status (SECURE_STATUS) Register" on page 16-55

Reset and Booting Registers

Table A-18. Reset and Booting Registers

Memory Mapped Address	Register Name	Description
0xFFC0 0100	SWRST	"Software Reset Register" on page 17-100
0xFFC0 0104	SYSCR	"System Reset Configuration Register" on page 17-102
0xEF00 0040	BK_REVISION	"Boot Code Revision Code (BK_REVISION)" on page 17-103
0xEF00 0050	BK_DATECODE	"Boot Code Date Code (BK_DATECODE)" on page 17-104
0xEF00 0048	BK_ZEROS	"Zero Word (BK_ZEROS)" on page 17-105
0xEF00 004C	BK_ONES	"Ones Word (BK_ONES)" on page 17-106
OTP 0x018L+(4 x i)		"Lower PBS00 Half Page (PBS00L, Bits 63–48)" on page 17-107
OTP 0x18H+(4 x i)		"Upper PBS00 Half Page (PBS00H, Bits 63–32)" on page 17-110
	PBS01L	not used
OTP 0x19H+(4xi)	PBS01H	"OTP Half Page (PBS01H, Bits 63–16)" on page 17-112
OTP 0x1AL+(4xi)	PBS02L	"Lower PBS02 Half Page (PBS02L Bits 63–0)" on page 17-114

Table A-18. Reset and Booting Registers (Continued)

Memory Mapped Address	Register Name	Description
	PBS02H	not used
	dFlags	"dFlags Word (Bits 31–16)" on page 17-120

TWI Registers

TWI registers (0x – 0x) are listed in Table A-19.

Table A-19. TWI Registers

Memory Mapped Address	Register Name	Description
0xFFC0 1404	TWI_CONTROL	"TWI Control Register" on page 23-27
0xFFC0 1400	TWI_CLKDIV	"SCL Clock Divider Register" on page 23-28
0xFFC0 1408	TWI_SLAVE_CTL	"TWI Slave Mode Control Register" on page 23-28
0xFFC0 1410	TWI_SLAVE_ADDR	"TWI Slave Mode Address Register" on page 23-30
0xFFC0 140C	TWI_SLAVE_STAT	"TWI Slave Mode Status Register" on page 23-30
0xFFC0 1414	TWI_MASTER_CTL	"TWI Master Mode Control Register" on page 23-32
0xFFC0 141C	TWI_MASTER_ADDR	"TWI Master Mode Address Register" on page 23-35
0xFFC0 1418	TWI_MASTER_STAT	"TWI Master Mode Status Register" on page 23-35
0xFFC0 1428	TWI_FIFO_CTL	"TWI FIFO Control Register" on page 23-38
0xFFC0 142C	TWI_FIFO_STAT	"TWI FIFO Status Register" on page 23-40
0xFFC0 1424	TWI_INT_MASK	"TWI Interrupt Mask Register" on page 23-42
0xFFC0 1420	TWI_INT_STAT	"TWI Interrupt Status Register" on page 23-42
0xFFC0 1480	TWI_XMT_DATA8	"TWI FIFO Transmit Data Single Byte Register" on page 23-45
0xFFC0 1484	TWI_XMT_DATA16	"TWI FIFO Transmit Data Double Byte Register" on page 23-46
0xFFC0 1488	TWI_RCV_DATA8	"TWI FIFO Receive Data Single Byte Register" on page 23-47
0xFFC0 148C	TWI_RCV_DATA16	"TWI FIFO Receive Data Double Byte Register" on page 23-48

UART1 Controller Registers

UART1 controller registers (0xFFC0 2000 – 0xFFC0 20FF) are listed in Table A-20. For UART0 registers, see Table A-6 on page A-6.

Table A-20. UART1 Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 2000	UART1_THR	"UART Transmit Holding Register" on page 25-27
0xFFC0 2000	UART1_RBR	"UART Receive Buffer Register" on page 25-27
0xFFC0 2000	UART1_DLL	"UART Divisor Latch Registers" on page 25-31
0xFFC0 2004	UART1_DLH	"UART Divisor Latch Registers" on page 25-31
0xFFC0 2004	UART1_IER	"UART Interrupt Enable Register" on page 25-28
0xFFC0 2008	UART1_IIR	"UART Interrupt Identification Register" on page 25-30
0xFFC0 200C	UART1_LCR	"UART Line Control Register" on page 25-22
0xFFC0 2010	UART1_MCR	"UART Modem Control Registers" on page 25-24
0xFFC0 2014	UART1_LSR	"UART Line Status Register" on page 25-25
0xFFC0 201C	UART1_SCR	"UART Scratch Register" on page 25-32
0xFFC0 2024	UART1_GCTL	"UART Global Control Register" on page 25-32

Ethernet MAC Registers

Ethernet MAC registers (0xFFC0 3000 – 0xFFC0 31FF) are listed in Table A-21.

Table A-21. Ethernet MAC Registers

Memory Mapped Address	Register Name	Description
0xFFC0 3000	EMAC_OPMODE	"EMAC_OPMODE Register" on page 21-61
0xFFC0 3004	EMAC_ADDRLO	"EMAC_ADDRLO Register" on page 21-68

Table A-21. Ethernet MAC Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 3008	EMAC_ADDRHI	"EMAC_ADDRHI Register" on page 21-68
0xFFC0 300C	EMAC_HASHLO	"EMAC_HASHLO Register" on page 21-71
0xFFC0 3010	EMAC_HASHHI	"EMAC_HASHHI Register" on page 21-72
0xFFC0 3014	EMAC_STAADD	"EMAC_STAADD Register" on page 21-73
0xFFC0 3018	EMAC_STADAT	"EMAC_STADAT Register" on page 21-74
0xFFC0 301C	EMAC_FLC	"EMAC_FLC Register" on page 21-75
0xFFC0 3020	EMAC_VLAN1	"EMAC_VLAN1 Register" on page 21-77
0xFFC0 3024	EMAC_VLAN2	"EMAC_VLAN2 Register" on page 21-78
0xFFC0 302C	EMAC_WKUP_CTL	"EMAC_WKUP_CTL Register" on page 21-79
0xFFC0 3030	EMAC_WKUP_FFMSK0	"EMAC_WKUP_FFMSK0 Register" on page 21-82
0xFFC0 3034	EMAC_WKUP_FFMSK1	"EMAC_WKUP_FFMSK1 Register" on page 21-83
0xFFC0 3038	EMAC_WKUP_FFMSK2	"EMAC_WKUP_FFMSK2 Register" on page 21-84
0xFFC0 303C	EMAC_WKUP_FFMSK3	"EMAC_WKUP_FFMSK3 Register" on page 21-85
0xFFC0 3040	EMAC_WKUP_FFCMD	"EMAC_WKUP_FFCMD Register" on page 21-86
0xFFC0 3044	EMAC_WKUP_FFOFF	"EMAC_WKUP_FFOFF Register" on page 21-88
0xFFC0 3048	EMAC_WKUP_FFCRC0/1	"EMAC_WKUP_FFCRC0 Register" on page 21-89
0xFFC0 304C	EMAC_WKUP_FFCRC2/3	"EMAC_WKUP_FFCRC0 Register" on page 21-89
0xFFC0 3060	EMAC_SYSCTL	"EMAC_SYSCTL Register" on page 21-90
0xFFC0 3064	EMAC_SYSTAT	"EMAC_SYSTAT Register" on page 21-92
0xFFC0 3068	EMAC_RX_STAT	"EMAC_RX_STAT Register" on page 21-95
0xFFC0 306C	EMAC_RX_STKY	"EMAC_RX_STKY Register" on page 21-100
0xFFC0 3070	EMAC_RX_IRQE	"EMAC_RX_IRQE Register" on page 21-104
0xFFC0 3074	EMAC_TX_STAT	"EMAC_TX_STAT Register" on page 21-105
0xFFC0 3078	EMAC_TX_STKY	"EMAC_TX_STKY Register" on page 21-109
0xFFC0 307C	EMAC_TX_IRQE	"EMAC_TX_IRQE Register" on page 21-112
0xFFC0 3080	EMAC_MMC_CTL	"EMAC_MMC_CTL Register" on page 21-121
0xFFC0 3084	EMAC_MMC_RIRQS	"EMAC_MMC_RIRQS Register" on page 21-113
0xFFC0 3088	EMAC_MMC_RIRQE	"EMAC_MMC_RIRQE Register" on page 21-115

Table A-21. Ethernet MAC Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 308C	EMAC_MMC_TIRQS	"EMAC_MMC_TIRQS Register" on page 21-117
0xFFC0 3090	EMAC_MMC_TIRQE	"EMAC_MMC_TIRQE Register" on page 21-119
0xFFC0 3100	EMAC_RXC_OK	"MAC Management Counter Registers" on page 21-54
0xFFC0 3104	EMAC_RXC_FCS	"MAC Management Counter Registers" on page 21-54
0xFFC0 3108	EMAC_RXC_ALIGN	"MAC Management Counter Registers" on page 21-54
0xFFC0 310C	EMAC_RXC_OCTET	"MAC Management Counter Registers" on page 21-54
0xFFC0 3110	EMAC_RXC_DMAOVF	"MAC Management Counter Registers" on page 21-54
0xFFC0 3114	EMAC_RXC_UNICST	"MAC Management Counter Registers" on page 21-54
0xFFC0 3118	EMAC_RXC_MULTI	"MAC Management Counter Registers" on page 21-54
0xFFC0 311C	EMAC_RXC_BROAD	"MAC Management Counter Registers" on page 21-54
0xFFC0 3120	EMAC_RXC_LNERRI	"MAC Management Counter Registers" on page 21-54
0xFFC0 3124	EMAC_RXC_LNERRO	"MAC Management Counter Registers" on page 21-54
0xFFC0 3128	EMAC_RXC_LONG	"MAC Management Counter Registers" on page 21-54
0xFFC0 312C	EMAC_RXC_MACCTL	"MAC Management Counter Registers" on page 21-54
0xFFC0 3130	EMAC_RXC_OPCODE	"MAC Management Counter Registers" on page 21-54
0xFFC0 3134	EMAC_RXC_PAUSE	"MAC Management Counter Registers" on page 21-54
0xFFC0 3138	EMAC_RXC_ALLFRM	"MAC Management Counter Registers" on page 21-54
0xFFC0 313C	EMAC_RXC_ALLOCT	"MAC Management Counter Registers" on page 21-54
0xFFC0 3140	EMAC_RXC_TYPED	"MAC Management Counter Registers" on page 21-54
0xFFC0 3144	EMAC_RXC_SHORT	"MAC Management Counter Registers" on page 21-54
0xFFC0 3148	EMAC_RXC_EQ64	"MAC Management Counter Registers" on page 21-54
0xFFC0 314C	EMAC_RXC_LT128	"MAC Management Counter Registers" on page 21-54
0xFFC0 3150	EMAC_RXC_LT256	"MAC Management Counter Registers" on page 21-54
0xFFC0 3154	EMAC_RXC_LT512	"MAC Management Counter Registers" on page 21-54
0xFFC0 3158	EMAC_RXC_LT1024	"MAC Management Counter Registers" on page 21-54
0xFFC0 315C	EMAC_RXC_GE1024	"MAC Management Counter Registers" on page 21-54
0xFFC0 3180	EMAC_TXC_OK	"MAC Management Counter Registers" on page 21-54
0xFFC0 3184	EMAC_TXC_1COL	"MAC Management Counter Registers" on page 21-54

Table A-21. Ethernet MAC Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 3188	EMAC_TXC_GT1COL	"MAC Management Counter Registers" on page 21-54
0xFFC0 318C	EMAC_TXC_OCTET	"MAC Management Counter Registers" on page 21-54
0xFFC0 3190	EMAC_TXC_DEFER	"MAC Management Counter Registers" on page 21-54
0xFFC0 3194	EMAC_TXC_LATECL	"MAC Management Counter Registers" on page 21-54
0xFFC0 3198	EMAC_TXC_XS_COL	"MAC Management Counter Registers" on page 21-54
0xFFC0 319C	EMAC_TXC_DMAUND	"MAC Management Counter Registers" on page 21-54
0xFFC0 31A0	EMAC_TXC_CRSERR	"MAC Management Counter Registers" on page 21-54
0xFFC0 31A4	EMAC_TXC_UNICST	"MAC Management Counter Registers" on page 21-54
0xFFC0 31A8	EMAC_TXC_MULTI	"MAC Management Counter Registers" on page 21-54
0xFFC0 31AC	EMAC_TXC_BROAD	"MAC Management Counter Registers" on page 21-54
0xFFC0 31B0	EMAC_TXC_ES_DFR	"MAC Management Counter Registers" on page 21-54
0xFFC0 31B4	EMAC_TXC_MACCTL	"MAC Management Counter Registers" on page 21-54
0xFFC0 31B8	EMAC_TXC_ALLFRM	"MAC Management Counter Registers" on page 21-54
0xFFC0 31BC	EMAC_TXC_ALLOCT	"MAC Management Counter Registers" on page 21-54
0xFFC0 31C0	EMAC_TXC_EQ64	"MAC Management Counter Registers" on page 21-54
0xFFC0 31C4	EMAC_TXC_LT128	"MAC Management Counter Registers" on page 21-54
0xFFC0 31C8	EMAC_TXC_LT254	"MAC Management Counter Registers" on page 21-54
0xFFC0 31CC	EMAC_TXC_LT512	"MAC Management Counter Registers" on page 21-54
0xFFC0 31D0	EMAC_TXC_LT1024	"MAC Management Counter Registers" on page 21-54
0xFFC0 31D4	EMAC_TXC_GE1024	"MAC Management Counter Registers" on page 21-54
0xFFC0 31D8	EMAC_TXC_ABORT	"MAC Management Counter Registers" on page 21-54

Handshake MDMA Control Registers

HMDMA registers (0xFFC0 3300 – 0xFFC0 33FF) are listed in Table A-22.

Table A-22. HMDMA Registers

Memory Mapped Address	Register Name	Description
0xFFC0 3300	HMDMA0_CONTROL	"Handshake MDMA Control Registers" on page 6-86
0xFFC0 3304	HMDMA0_ECINIT	"Handshake MDMA Initial Edge Count Registers" on page 6-89
0xFFC0 3308	HMDMA0_BCINIT	"Handshake MDMA Initial Block Count Registers" on page 6-87
0xFFC0 330C	HMDMA0_ECURGENT	"Handshake MDMA Edge Count Urgent Registers" on page 6-90
0xFFC0 3310	HMDMA0_ECOVERFLOW	"Handshake MDMA Edge Count Overflow Interrupt Registers" on page 6-90
0xFFC0 3314	HMDMA0_ECOUNT	"Handshake MDMA Current Edge Count Registers" on page 6-89
0xFFC0 3318	HMDMA0_BCOUNT	"Handshake MDMA Current Block Count Registers" on page 6-88
0xFFC0 3340	HMDMA1_CONTROL	"Handshake MDMA Control Registers" on page 6-86
0xFFC0 3344	HMDMA1_ECINIT	"Handshake MDMA Initial Edge Count Registers" on page 6-89
0xFFC0 3348	HMDMA1_BCINIT	"Handshake MDMA Initial Block Count Registers" on page 6-87
0xFFC0 334C	HMDMA1_ ECURGENT	"Handshake MDMA Edge Count Urgent Registers" on page 6-90
0xFFC0 3350	HMDMA1_ECOVERFLOW	"Handshake MDMA Edge Count Overflow Interrupt Registers" on page 6-90
0xFFC0 3354	HMDMA1_ECOUNT	"Handshake MDMA Current Edge Count Registers" on page 6-89
0xFFC0 3358	HMDMA1_BCOUNT	"Handshake MDMA Current Block Count Registers" on page 6-88

HOST DMA Port Registers

HOSTDP registers (0xFFC0 3400 – 0xFFC0 3408) are listed in Table A-23.

Table A-23. HOSTDP Registers

Memory Mapped Address	Register Name	Description
	HOST_CONFIG	"HOSTDP Configuration Word" on page 8-6
0xFFC0 3400	HOST_CONTROL	"HOSTDP Control Register" on page 8-26
0xFFC0 3404	HOST_STATUS	"HOSTDP Status Register" on page 8-29
0xFFC0 3408	HOST_TIMEOUT	"HOSTDP Timeout Register" on page 8-31

GP Counter Registers

GP Counter registers (0xFFC0 3500 – 0xFFC0 351C) are listed in Table A-24.

Table A-24. Rotary Registers

Memory Mapped Address	Register Name	Description
0xFFC0 3500	CNT_CONFIG	"Counter Configuration Register" on page 13-20
0xFFC0 3504	CNT_IMASK	"Counter Interrupt Mask Register" on page 13-21
0xFFC0 3508	CNT_STATUS	"Counter Status Register" on page 13-22
0xFFC0 350C	CNT_COMMAND	"Counter Command Register" on page 13-24
0xFFC0 3510	CNT_DEBOUNCE	"Counter Debounce Register" on page 13-25
0xFFC0 3514	CNT_COUNTER	"Counter Count Value Register" on page 13-26
0xFFC0 3518	CNT_MAX	"Counter Maximal Count Register" on page 13-27
0xFFC0 351C	CNT_MIN	"Counter Minimal Count Register" on page 13-27

NFC Registers

NFC registers (0xFFC0 3700 – 0xFFC0 37FC) are listed in Table A-25.

Table A-25. NFC Controller Registers

Memory Mapped Address	Register Name	Description
0xFFC0 3700	NFC_CTL	"NFC Control Register" on page 20-16
0xFFC0 3704	NFC_STAT	"NFC Status Register" on page 20-17
0xFFC0 3708	NFC_IRQSTAT	"NFC Interrupt Status Register" on page 20-18
0xFFC0370C	NFC_IRQMASK	"NFC Interrupt Mask Register" on page 20-19
0xFFC0 3710	NFC_ECC0	"NFC ECC Registers" on page 20-20
0xFFC0 3714	NFC_ECC1	"NFC ECC Registers" on page 20-20
0xFFC0 3718	NFC_ECC2	"NFC ECC Registers" on page 20-20
0xFFC0 371C	NFC_ECC3	"NFC ECC Registers" on page 20-20
0xFFC0 3720	NFC_COUNT	"NFC Count Register" on page 20-21
0xFFC0 3724	NFC_RST	"NFC Reset Register" on page 20-21
0xFFC0 3728	NFC_PGCTL	"NFC Page Control Register" on page 20-22
0xFFC0 372C	NFC_READ	"NFC Read Data Register" on page 20-23
0xFFC0 3740	NFC_ADDR	"NFC Address Register" on page 20-23
0xFFC0 3744	NFC_CMD	"NFC Command Register" on page 20-24
0xFFC0 3748	NFC_DATA_WR	"NFC Data Write Register" on page 20-24
0xFFC0 374C	NFC_DATA_RD	"NFC Data Read Register" on page 20-25

USB Registers

USB registers (0xFFC0 3800 – 0xFFC0 3Cxx) are listed in Table A-26.

Table A-26. USB Registers

Memory Mapped Address	Register Name	Description
0xFFC0 3800	USB_FADDR	"USB Function Address Register" on page 26-100
0xFFC0 3804	USB_POWER	"USB Power Management Register" on page 26-98
0xFFC0 3808	USB_INTRTX	"USB Transmit Interrupt Register" on page 26-103
0xFFC0 380C	USB_INTRRX	"USB Receive Interrupt Register" on page 26-104
0xFFC0 3810	USB_INTRTXE	"USB Transmit Interrupt Enable Register" on page 26-105
0xFFC0 3814	USB_INTRRXE	"USB Receive Interrupt Enable Register" on page 26-106
0xFFC0 3818	USB_INTRUSB	"USB Common Interrupts Register" on page 26-107
0xFFC0 381C	USB_INTRUSBE	"USB Common Interrupts Enable Register" on page 26-108
0xFFC0 3820	USB_FRAME	"USB Frame Number Register" on page 26-109
0xFFC0 3824	USB_INDEX	"USB Index Register" on page 26-109
0xFFC0 3828	USB_TESTMODE	"USB Test Mode Register" on page 26-101
0xFFC0 382C	USB_GLOBINTR	"USB Global Interrupt Register" on page 26-102
0xFFC0 3830	USB_GLOBAL_CTL	"USB Global Control Register" on page 26-96
0xFFC0 3840	USB_TX_MAX_PACKET	"USB TX Max Packet Register" on page 26-110
0xFFC0 3844	USB_CSR0	"USB Control/Status EP0 Register" on page 26-111
0xFFC0 3844	USB_TXCSR	"USB TX Control/Status EPx Register" on page 26-115
0xFFC0 3848	USB_RX_MAX_PACKET	"USB RX Max Packet Register" on page 26-120
0xFFC0 384C	USB_RXCSR	"USB RX Control/Status EPx Register" on page 26-121
0xFFC0 3850	USB_COUNT0	"USB Transmit Interrupt Register" on page 26-103
0xFFC0 3850	USB_RXCOUNT	"USB RX Byte Count Register" on page 26-127
0xFFC0 3854	USB_TXTYPE	"USB TX Type Register" on page 26-127

Table A-26. USB Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 3858	USB_NAKLIMIT0	"USB NAK Limit 0 Register" on page 26-128
0xFFC0 3858	USB_TXINTERVAL	"USB TX Interval Register" on page 26-128
0xFFC0 385C	USB_RXTYPE	"USB RX Type Register" on page 26-129
0xFFC0 3860	USB_RXINTERVAL	"USB RX Interval Register" on page 26-130
0xFFC0 3868	USB_TXCOUNT	"USB TX Byte Count EPx Register" on page 26-131
0xFFC0 3880	USB_EP0_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-132
0xFFC0 3888	USB_EP1_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-132
0xFFC0 3890	USB_EP2_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-132
0xFFC0 3898	USB_EP3_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-132
0xFFC0 38A0	USB_EP4_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-132
0xFFC0 38A8	USB_EP5_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-132
0xFFC0 38B0	USB_EP6_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-132
0xFFC0 38B8	USB_EP7_FIFO	"USB Endpoint FIFO (USB_EPx_FIFO) Registers" on page 26-132
0xFFC0 3900	USB_OTG_DEV_CTL	"USB OTG Device Control Register" on page 26-132
0xFFC0 3904	USB_OTG_VBUS_IRQ	"USB OTG VBUS Interrupt Register" on page 26-134
0xFFC0 3908	USB_OTG_VBUS_MASK	"USB OTG VBUS Mask Register" on page 26-136
0xFFC0 3948	USB_LINKINFO	"USB Link Info Register" on page 26-137
0xFFC0 394C	USB_VPLEN	"USB VBUS Pulse Length Register" on page 26-137
0xFFC0 3950	USB_HS_EOF1	"USB High-Speed EOF 1 Register" on page 26-138
0xFFC0 3954	USB_FS_EOF1	"USB Full-Speed EOF 1 Register" on page 26-138
0xFFC0 3958	USB_LS_EOF1	"USB Low-Speed EOF 1 Register" on page 26-139

Table A-26. USB Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 39E8	USB_APHY_CNTRL2	"USB APHY Control 2 Register" on page 26-140
0xFFC0 39F0	USB_PLLOSC_CTRL	"USB PLL OSC Control Register" on page 26-141
0xFFC0 39F4	USB_SRP_CLKDIV	"USB SRP Clock Divider Register" on page 26-142
0xFFC0 3A04	USB_EP_NI0_TXCSR	Control status register for endpoint 0
0xFFC0 3A0C	USB_EP_NI0_RXCSR	Control status register for host RX endpoint 0
0xFFC0 3A10	USB_EP_NI0_RXCOUNT	Number of bytes received in endpoint 0 FIFO
0xFFC0 3A14	USB_EP_NI0_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host TX endpoint 0
0xFFC0 3A18	USB_EP_NI0_TXINTERVAL	Sets the NAK response timeout on endpoint 0
0xFFC0 3A1C	USB_EP_NI0_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host RX endpoint 0
0xFFC0 3A20	USB_EP_NI0_RXINTERVAL	Sets the polling interval for interrupt/isochronous transfers or the NAK response timeout on bulk transfers for host RX endpoint 0
0xFFC0 3A28	USB_EP_NI0_TXCOUNT	Number of bytes to be written to the endpoint 0 TX FIFO
0xFFC0 3A40	USB_EP_NI1_TXMAXP	Maximum packet size for Host TX endpoint 1
0xFFC0 3A44	USB_EP_NI1_TXCSR	Control status register for endpoint 1
0xFFC0 3A48	USB_EP_NI1_RXMAXP	Maximum packet size for host RX endpoint 1
0xFFC0 3A4C	USB_EP_NI1_RXCSR	Control status register for host RX endpoint 1
0xFFC0 3A50	USB_EP_NI1_RXCOUNT	Number of bytes received in endpoint 1 FIFO
0xFFC0 3A54	USB_EP_NI1_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host TX endpoint 1
0xFFC0 3A58	USB_EP_NI1_TXINTERVAL	Sets the NAK response timeout on endpoint 1
0xFFC0 3A5C	USB_EP_NI1_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host RX endpoint 1
0xFFC0 3A60	USB_EP_NI1_RXINTERVAL	Sets the polling interval for interrupt/isochronous transfers or the NAK response timeout on bulk transfers for host RX endpoint 1
0xFFC0 3A68	USB_EP_NI1_TXCOUNT	Number of bytes to be written to the endpoint 1 TX FIFO

Table A-26. USB Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 3A80	USB_EP_NI2_TXMAXP	Maximum packet size for Host TX endpoint 2
0xFFC0 3A84	USB_EP_NI2_TXCSR	Control status register for endpoint 2
0xFFC0 3A88	USB_EP_NI2_RXMAXP	Maximum packet size for host RX endpoint 2
0xFFC0 3A8C	USB_EP_NI2_RXCSR	Control status register for host RX endpoint 2
0xFFC0 3A90	USB_EP_NI2_RXCOUNT	Number of bytes received in endpoint 2 FIFO
0xFFC0 3A94	USB_EP_NI2_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host TX endpoint 2
0xFFC0 3A98	USB_EP_NI2_TXINTERVAL	Sets the NAK response timeout on endpoint 2
0xFFC0 3A9C	USB_EP_NI2_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host RX endpoint 2
0xFFC0 3AA0	USB_EP_NI2_RXINTERVAL	Sets the polling interval for interrupt/isochronous transfers or the NAK response timeout on bulk transfers for host RX endpoint 2
0xFFC0 3AA8	USB_EP_NI2_TXCOUNT	Number of bytes to be written to the endpoint 2 TX FIFO
0xFFC0 3AC0	USB_EP_NI3_TXMAXP	Maximum packet size for Host TX endpoint 3
0xFFC0 3AC4	USB_EP_NI3_TXCSR	Control status register for endpoint 3
0xFFC0 3AC8	USB_EP_NI3_RXMAXP	Maximum packet size for host RX endpoint 3
0xFFC0 3ACC	USB_EP_NI3_RXCSR	Control status register for host RX endpoint 3
0xFFC0 3AD0	USB_EP_NI3_RXCOUNT	Number of bytes received in endpoint 3 FIFO
0xFFC0 3AD4	USB_EP_NI3_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host TX endpoint 3
0xFFC0 3AD8	USB_EP_NI3_TXINTERVAL	Sets the NAK response timeout on endpoint 3
0xFFC0 3ADC	USB_EP_NI3_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host RX endpoint 3
0xFFC0 3AE0	USB_EP_NI3_RXINTERVAL	Sets the polling interval for interrupt/isochronous transfers or the NAK response timeout on bulk transfers for host RX endpoint 3
0xFFC0 3AE8	USB_EP_NI3_TXCOUNT	Number of bytes to be written to the endpoint 3 TX FIFO
0xFFC0 3B00	USB_EP_NI4_TXMAXP	Maximum packet size for Host TX endpoint 4

Table A-26. USB Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 3B04	USB_EP_NI4_TXCSR	Control status register for endpoint 4
0xFFC0 3B08	USB_EP_NI4_RXMAXP	Maximum packet size for host RX endpoint 4
0xFFC0 3B0C	USB_EP_NI4_RXCSR	Control status register for host RX endpoint 4
0xFFC0 3B10	USB_EP_NI4_RXCOUNT	Number of bytes received in endpoint 4 FIFO
0xFFC0 3B14	USB_EP_NI4_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host TX endpoint 4
0xFFC0 3B18	USB_EP_NI4_TXINTERVAL	Sets the NAK response timeout on endpoint 4
0xFFC0 3B1C	USB_EP_NI4_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host RX endpoint 4
0xFFC0 3B20	USB_EP_NI4_RXINTERVAL	Sets the polling interval for interrupt/isochronous transfers or the NAK response timeout on bulk transfers for host RX endpoint 4
0xFFC0 3B28	USB_EP_NI4_TXCOUNT	Number of bytes to be written to the endpoint 4 TX FIFO
0xFFC0 3B40	USB_EP_NI5_TXMAXP	Maximum packet size for Host TX endpoint 5
0xFFC0 3B44	USB_EP_NI5_TXCSR	Control status register for endpoint 5
0xFFC0 3B48	USB_EP_NI5_RXMAXP	Maximum packet size for host RX endpoint 5
0xFFC0 3B4C	USB_EP_NI5_RXCSR	Control status register for host RX endpoint 5
0xFFC0 3B50	USB_EP_NI5_RXCOUNT	Number of bytes received in endpoint 5 FIFO
0xFFC0 3B54	USB_EP_NI5_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host TX endpoint 5
0xFFC0 3B58	USB_EP_NI5_TXINTERVAL	Sets the NAK response timeout on endpoint 5
0xFFC0 3B5C	USB_EP_NI5_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host RX endpoint 5
0xFFC0 3B60	USB_EP_NI5_RXINTERVAL	Sets the polling interval for interrupt/isochronous transfers or the NAK response timeout on bulk transfers for host RX endpoint 5
0xFFC0 3B68	USB_EP_NI5_TXCOUNT	Number of bytes to be written to the endpoint 5 TX FIFO
0xFFC0 3B80	USB_EP_NI6_TXMAXP	Maximum packet size for Host TX endpoint 6
0xFFC0 3B84	USB_EP_NI6_TXCSR	Control status register for endpoint 6

Table A-26. USB Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 3B88	USB_EP_NI6_RXMAXP	Maximum packet size for host RX endpoint 6
0xFFC0 3B8C	USB_EP_NI6_RXCSR	Control status register for host RX endpoint 6
0xFFC0 3B90	USB_EP_NI6_RXCOUNT	Number of bytes received in endpoint 6 FIFO
0xFFC0 3B94	USB_EP_NI6_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host TX endpoint 6
0xFFC0 3B98	USB_EP_NI6_TXINTERVAL	Sets the NAK response timeout on endpoint 6
0xFFC0 3B9C	USB_EP_NI6_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host RX endpoint 6
0xFFC0 3BA0	USB_EP_NI6_RXINTERVAL	Sets the polling interval for interrupt/isochronous transfers or the NAK response timeout on bulk transfers for host RX endpoint 6
0xFFC0 3BA8	USB_EP_NI6_TXCOUNT	Number of bytes to be written to the endpoint 6 TX FIFO
0xFFC0 3BC0	USB_EP_NI7_TXMAXP	Maximum packet size for Host TX endpoint 7
0xFFC0 3BC4	USB_EP_NI7_TXCSR	Control status register for endpoint 7
0xFFC0 3BC8	USB_EP_NI7_RXMAXP	Maximum packet size for host RX endpoint 7
0xFFC0 3BCC	USB_EP_NI7_RXCSR	Control status register for host RX endpoint 7
0xFFC0 3BD0	USB_EP_NI7_RXCOUNT	Number of bytes received in endpoint 7 FIFO
0xFFC0 3BD4	USB_EP_NI7_TXTYPE	Sets the transaction protocol and peripheral endpoint number for the host TX endpoint 7
0xFFC0 3BD8	USB_EP_NI7_TXINTERVAL	Sets the NAK response timeout on endpoint 7
0xFFC0 3BDC	USB_EP_NI7_RXTYPE	Sets the transaction protocol and peripheral endpoint number for the host RX endpoint 7
0xFFC0 3BF0	USB_EP_NI7_RXINTERVAL	Sets the polling interval for interrupt/isochronous transfers or the NAK response timeout on bulk transfers for host RX endpoint 7
0xFFC0 3BF8	USB_EP_NI7_TXCOUNT	Number of bytes to be written to the endpoint 7 TX FIFO
0xFFC0 3C00	USB_DMA_INTERRUPT	“USB DMA Interrupt Register” on page 26-143
0xFFC0 3C04	USB_DMA0CONTROL	“USB DMAx Control Registers” on page 26-144

Table A-26. USB Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 3C08	USB_DMA0ADDRLOW	"USB DMAx Address Low Registers" on page 26-146
0xFFC0 3C0C	USB_DMA0ADDRHIGH	"USB DMAx Address High Registers" on page 26-146
0xFFC0 3C10	USB_DMA0COUNTLOW	"USB DMAx Count Low Registers" on page 26-147
0xFFC0 3C14	USB_DMA0COUNTHIGH	"USB DMAx Count High Registers" on page 26-148
0xFFC0 3C24	USB_DMA1CONTROL	"USB DMAx Control Registers" on page 26-144
0xFFC0 3C28	USB_DMA1ADDRLOW	"USB DMAx Address Low Registers" on page 26-146
0xFFC0 3C2C	USB_DMA1ADDRHIGH	"USB DMAx Address High Registers" on page 26-146
0xFFC0 3C30	USB_DMA1COUNTLOW	"USB DMAx Count Low Registers" on page 26-147
0xFFC0 3C34	USB_DMA1COUNTHIGH	"USB DMAx Count High Registers" on page 26-148
0xFFC0 3C44	USB_DMA2CONTROL	"USB DMAx Control Registers" on page 26-144
0xFFC0 3C48	USB_DMA2ADDRLOW	"USB DMAx Address Low Registers" on page 26-146
0xFFC0 3C4C	USB_DMA2ADDRHIGH	"USB DMAx Address High Registers" on page 26-146
0xFFC0 3C50	USB_DMA2COUNTLOW	"USB DMAx Count Low Registers" on page 26-147
0xFFC0 3C54	USB_DMA2COUNTHIGH	"USB DMAx Count High Registers" on page 26-148
0xFFC0 3C64	USB_DMA3CONTROL	"USB DMAx Control Registers" on page 26-144
0xFFC0 3C68	USB_DMA3ADDRLOW	"USB DMAx Address Low Registers" on page 26-146
0xFFC0 3C6C	USB_DMA3ADDRHIGH	"USB DMAx Address High Registers" on page 26-146
0xFFC0 3C70	USB_DMA3COUNTLOW	"USB DMAx Count Low Registers" on page 26-147
0xFFC0 3C74	USB_DMA3COUNTHIGH	"USB DMAx Count High Registers" on page 26-148
0xFFC0 3C84	USB_DMA4CONTROL	"USB DMAx Control Registers" on page 26-144
0xFFC0 3C88	USB_DMA4ADDRLOW	"USB DMAx Address Low Registers" on page 26-146

Table A-26. USB Registers (Continued)

Memory Mapped Address	Register Name	Description
0xFFC0 3C8C	USB_DMA4ADDRHIGH	"USB DMAx Address High Registers" on page 26-146
0xFFC0 3C90	USB_DMA4COUNTLOW	"USB DMAx Count Low Registers" on page 26-147
0xFFC0 3C94	USB_DMA4COUNTHIGH	"USB DMAx Count High Registers" on page 26-148
0xFFC0 3CA4	USB_DMA5CONTROL	"USB DMAx Control Registers" on page 26-144
0xFFC0 3CA8	USB_DMA5ADDRLOW	"USB DMAx Address Low Registers" on page 26-146
0xFFC0 3CAC	USB_DMA5ADDRHIGH	"USB DMAx Address High Registers" on page 26-146
0xFFC0 3CB0	USB_DMA5COUNTLOW	"USB DMAx Count Low Registers" on page 26-147
0xFFC0 3CB4	USB_DMA5COUNTHIGH	"USB DMAx Count High Registers" on page 26-148
0xFFC0 3CC4	USB_DMA6CONTROL	"USB DMAx Control Registers" on page 26-144
0xFFC0 3CC8	USB_DMA6ADDRLOW	"USB DMAx Address Low Registers" on page 26-146
0xFFC0 3CCC	USB_DMA6ADDRHIGH	"USB DMAx Address High Registers" on page 26-146
0xFFC0 3CD0	USB_DMA6COUNTLOW	"USB DMAx Count Low Registers" on page 26-147
0xFFC0 3CD4	USB_DMA6COUNTHIGH	"USB DMAx Count High Registers" on page 26-148
0xFFC0 3CE4	USB_DMA7CONTROL	"USB DMAx Control Registers" on page 26-144
0xFFC0 3CE8	USB_DMA7ADDRLOW	"USB DMAx Address Low Registers" on page 26-146
0xFFC0 3CEC	USB_DMA7ADDRHIGH	"USB DMAx Address High Registers" on page 26-146
0xFFC0 3CF0	USB_DMA7COUNTLOW	"USB DMAx Count Low Registers" on page 26-147
0xFFC0 3CF4	USB_DMA7COUNTHIGH	"USB DMAx Count High Registers" on page 26-148

Processor-Specific Memory Registers

Processor-specific memory registers (0xFFE0 0004 – 0xFFE0 0300) are listed in Table A-27.

Table A-27. Processor-Specific Memory Registers

Memory Mapped Address	Register Name	Description
0xFFE0 0004	DMEM_CONTROL	"L1 Data Memory Control Register" on page 3-6
0xFFE0 0300	DTEST_COMMAND	"Data Test Command Register" on page 3-7

Core Timer Registers

Core timer registers (0xFFE0 3000 – 0xFFE0 300C) are listed in Table A-28.

Table A-28. Core Timer Registers

Memory Mapped Address	Register Name	Description
0xFFE0 3000	TCNTL	"Core Timer Control Register" on page 11-5
0xFFE0 3004	TPERIOD	"Core Timer Period Register" on page 11-6
0xFFE0 3008	TSCALE	"Core Timer Scale Register" on page 11-7
0xFFE0 300C	TCOUNT	"Core Timer Count Register" on page 11-6

B TEST FEATURES

This appendix discusses the test features of the processor.

JTAG Standard

The processor is fully compatible with the IEEE 1149.1 standard, also known as the Joint Test Action Group (JTAG) standard.

The JTAG standard defines circuitry that may be built to assist in the test, maintenance, and support of assembled printed circuit boards.The circuitry includes a standard interface through which instructions and test data are communicated. A set of test features is defined, including a boundary-scan register, such that the component can respond to a minimum set of instructions designed to help test printed circuit boards.

The standard defines test logic that can be included in an integrated circuit to provide standardized approaches to:

- Testing the interconnections between integrated circuits once they have been assembled onto a printed circuit board
- Testing the integrated circuit itself
- Observing or modifying circuit activity during normal component operation

The test logic consists of a boundary-scan register and other building blocks. The test logic is accessed through a Test Access Port (TAP).

Full details of the JTAG standard can be found in the document *IEEE Standard Test Access Port and Boundary-Scan Architecture*, ISBN 1-55937-350-4.

Boundary-Scan Architecture

The boundary-scan test logic consists of:

- A TAP comprised of five pins (see Table B-1)
- A TAP controller that controls all sequencing of events through the test registers
- An instruction register (IR) that interprets 5-bit instruction codes to select the test mode that performs the desired test operation
- Several data registers defined by the JTAG standard

Table B-1. Test Access Port Pins

Pin Name	Input/Output	Description
TDI	Input	Test Data Input
TMS	Input	Test Mode Select
TCK	Input	Test Clock
$\overline{\text{TRST}}$	Input	Test Reset
TDO	Output	Test Data Out

The TAP controller is a synchronous, 16-state, finite-state machine controlled by the TCK and TMS pins. Transitions to the various states in the diagram occur on the rising edge of TCK and are defined by the state of the TMS pin, here denoted by either a logic 1 or logic 0 state. For full details of the operation, see the JTAG standard.

Figure B-1 shows the state diagram for the TAP controller.

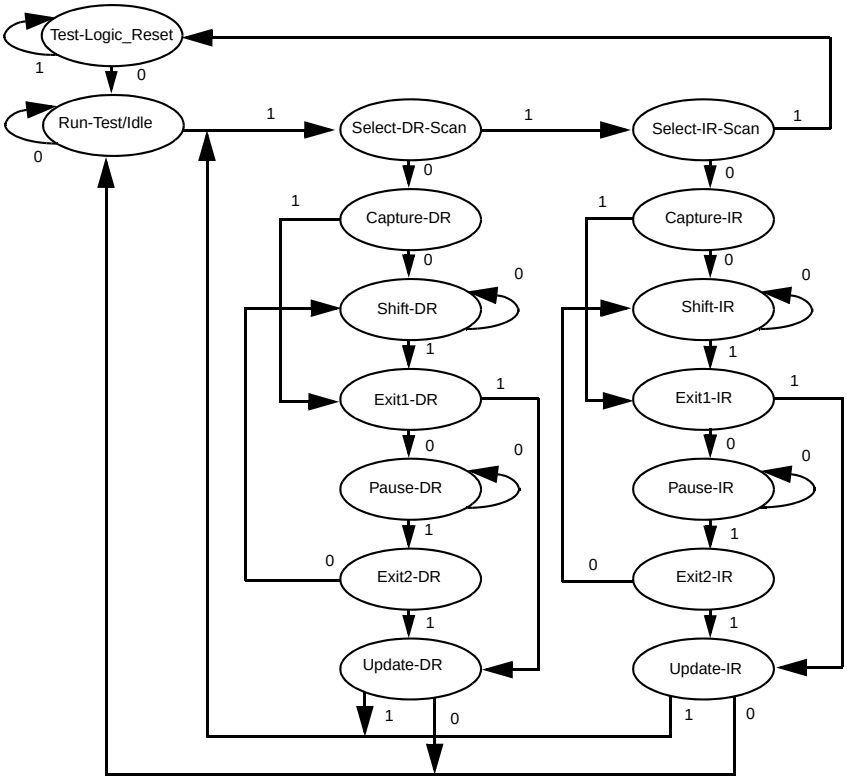


Figure B-1. TAP Controller State Diagram

Note:

- The TAP controller enters the test-logic-reset state when TMS is held high after five TCK cycles.
- The TAP controller enters the test-logic-reset state when TRST is asynchronously asserted.
- An external system reset does not affect the state of the TAP controller, nor does the state of the TAP controller affect an external system reset.

Instruction Register

The instruction register is five bits wide and accommodates up to 32 boundary-scan instructions.

The instruction register holds both public and private instructions. The JTAG standard requires some of the public instructions; other public instructions are optional. Private instructions are reserved for the manufacturer's use.

The binary decode column of Table B-2 lists the decode for the public instructions. The register column lists the serial scan paths.

Table B-2. Decode for Public JTAG-Scan Instructions

Instruction Name	Binary Decode 01234	Register
EXTEST	00000	Boundary-Scan
SAMPLE/PRELOAD	10000	Boundary-Scan
BYPASS	11111	Bypass

Figure B-2 shows the instruction bit scan ordering for the paths shown in Table B-2.

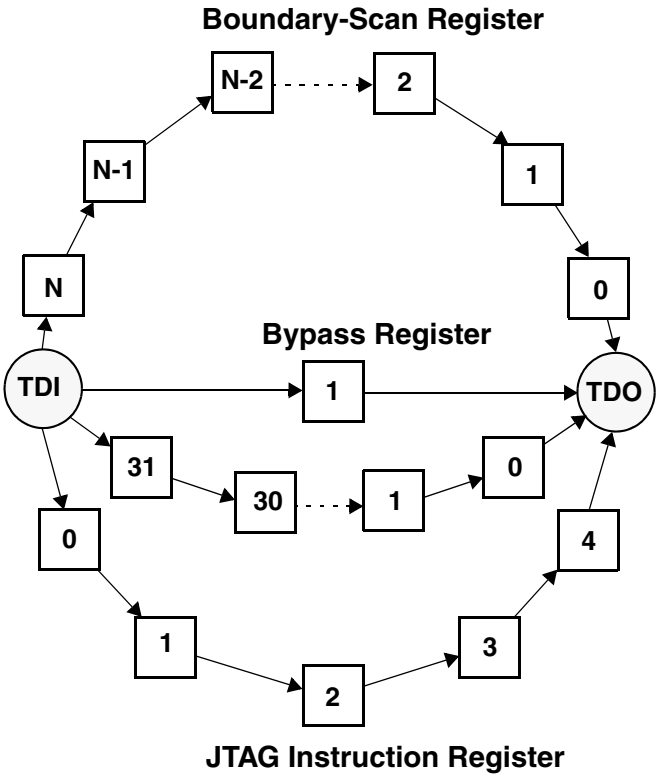


Figure B-2. Serial Scan Paths

Public Instructions

The following sections describe the public JTAG scan instructions.

EXTEST – Binary Code 00000

The EXTEST instruction selects the boundary-scan register to be connected between the TDI and TDO pins. This instruction allows testing of on-board circuitry external to the device.

The EXTEST instruction allows internal data to be driven to the boundary outputs and external data to be captured on the boundary inputs.

To protect the internal logic when the boundary outputs are overdriven or signals are received on the boundary inputs, make sure that nothing else drives data on the processor's output pins.

SAMPLE/PRELOAD – Binary Code 10000

The SAMPLE/PRELOAD instruction performs two functions and selects the Boundary-Scan register to be connected between TDI and TDO. The instruction has no effect on internal logic.

The SAMPLE part of the instruction allows a snapshot of the inputs and outputs captured on the boundary-scan cells. Data is sampled on the rising edge of TCK.

The PRELOAD part of the instruction allows data to be loaded on the device pins and driven out on the board with the EXTEST instruction. Data is preloaded on the pins on the falling edge of TCK.

BYPASS – Binary Code 11111

The BYPASS instruction selects the BYPASS register to be connected to TDI and TDO. The instruction has no effect on the internal logic. No data inversion should occur between TDI and TDO.

Boundary-Scan Register

The boundary-scan register is selected by the EXTEST and SAMPLE/PRELOAD instructions. These instructions allow the pins of the processor to be controlled and sampled for board-level testing.

G GLOSSARY

ALU.

See *Arithmetic/Logic Unit*

AMC (Asynchronous Memory Controller).

A configurable memory controller supporting multiple banks of asynchronous memory including SRAM, ROM, and flash, where each bank can be independently programmed with different timing parameters.

Arithmetic/Logic Unit (ALU).

A processor component that performs arithmetic, comparative, and logical functions.

bank activate command.

The bank activate command causes the SDRAM to open an internal bank (specified by the bank address) in a row (specified by the row address). When the bank activate command is issued, it opens a new row address in the dedicated bank. The memory in the open internal bank and row is referred to as the open page. The bank activate command must be applied before a read or write command.

base address.

The starting address of a circular buffer.

base register.

A Data Address Generator (DAG) register that contains the starting address for a circular buffer.

bit-reversed addressing.

The addressing mode in which the Data Address Generator (DAG) provides a bit-reversed address during a data move without reversing the stored address.

Boot memory space.

Internal memory space designated for a program that is executed immediately after powerup or after a software reset.

burst length.

The burst length determines the number of words that the SDRAM device stores or delivers after detecting a single write or read command followed by a NOP (no operation) command, respectively (`Number of NOPs = burst length - 1`). Burst lengths of full page, 8, 4, 2, and 1 (no burst) are available. The burst length is selected by writing the `BL` bits in the SDRAM's mode register during the SDRAM powerup sequence.

Burst Stop command.

The burst stop command is one of several ways to terminate a burst read or write operation.

burst type.

The burst type determines the address order in which the SDRAM delivers burst data. The burst type is selected by writing the `BT` bits in the SDRAM's mode register during the SDRAM powerup sequence.

cache block.

The smallest unit of memory that is transferred to/from the next level of memory from/to a cache as a result of a cache miss.

cache hit.

A memory access that is satisfied by a valid, present entry in the cache.

cache line.

Same as cache block. In this document, *cache line* is used for *cache block*.

cache miss.

A memory access that does not match any valid entry in the cache.

cache tag.

Upper address bits, stored along with the cached data line, to identify the specific address source in memory that the cached line represents.

Cacheability Protection Lookaside Buffer (CPLB).

Storage area that describes the access characteristics of the core memory map.

CAM (Content Addressable Memory).

Also called associative memory. A memory device that includes comparison logic with each bit of storage. A data value is broadcast to all words in memory; it is compared with the stored values; and values that match are flagged.

CAS (Column Address Strobe).

A signal sent from the SDC to a DRAM device to indicate that the column address lines are valid.

CAS latency (also t_{AA}, t_{CAC}, CL).

The CAS latency or read latency specifies the time between latching a read address and driving the data off chip. This specification is normalized to the system clock and varies from 2 to 3 cycles based on the speed. The CAS latency is selected by writing the `CL` bits in the SDRAM's mode register during the SDRAM powerup sequence.

CBR (CAS Before RAS) memory refresh.

DRAM devices have a built-in counter for the refresh row address. By activating Column Address Strobe (CAS) before activating Row Address Strobe (RAS), this counter is selected to supply the row address instead of the address inputs.

CEC.

See *Core Event Controller*

circular addressing.

The process by which the Data Address Generator (DAG) "wraps around" or repeatedly steps through a range of registers.

companding.

(Compressing/expanding). The process of logarithmically encoding and decoding data to minimize the number of bits that must be sent.

conditional branches.

Jump or call/return instructions whose execution is based on defined conditions.

core.

The core consists of these functional blocks: CPU, L1 memory, event controller, core timer, and performance monitoring registers.

Core Event Controller (CEC).

The CEC works with the System Interrupt Controller (SIC) to prioritize and control all system interrupts. The CEC handles general-purpose interrupts and interrupts routed from the SIC.

CPLB.

See *Cacheability Protection Lookaside Buffer*

DAB.

See *DMA Access Bus*

DAG.

See *Data Address Generator*

Data Address Generator (DAG).

Processing component that provides memory addresses when data is transferred between memory and registers.

Data Register File.

A set of data registers that is used to transfer data between computation units and memory while providing local storage for operands.

data registers (Dreg).

Registers located in the data arithmetic unit that hold operands and results for multiplier, ALU, or shifter operations.

DCB.

See *DMA Core Bus*

DEB.

See *DMA External Bus*

descriptor block, DMA.

A set of parameters used by the direct memory access (DMA) controller to describe a set of DMA sequences.

descriptor loading, DMA.

The process in which the direct memory access (DMA) controller downloads a DMA descriptor from data memory and autoinitializes the DMA parameter registers.

DFT (Design For Testability).

A set of techniques that helps designers of digital systems ensure that those systems will be testable.

Digital Signal Processor (DSP).

An integrated circuit designated for high-speed manipulation of analog information that has been converted into digital form.

direct branches.

Jump or call/return instructions that use absolute addresses that do not change at runtime (such as a program label), or they use a PC-relative address.

direct-mapped.

Cache architecture where each line has only one place that it can appear in the cache. Also described as 1-way associative.

Direct Memory Access (DMA).

A way of moving data between system devices and memory in which the data is transferred through a DMA port without involving the processor.

dirty, modified.

A state bit, stored along with the tag, indicating whether the data in the data cache line has been changed since it was copied from the source memory and, therefore, needs to be updated in that source memory.

DMA.

See *Direct Memory Access*

DMA Access Bus (DAB).

A bus that provides a means for DMA channels to be accessed by the peripherals.

DMA chaining.

The linking or chaining of multiple direct memory access (DMA) sequences. In chained DMA, the I/O processor loads the next DMA descriptor into the DMA parameter registers when the current DMA finishes and autoinitializes the next DMA sequence.

DMA Core Bus (DCB).

A bus that provides a means for DMA channels to gain access to on-chip memory.

DMA descriptor registers.

Registers that hold the initialization information for a direct memory access (DMA) process.

DMA External Bus (DEB).

A bus that provides a means for DMA channels to gain access to off-chip memory.

DPMC (Dynamic Power Management Controller).

A processor's control block that allows the user to dynamically control the processor's performance characteristics and power dissipation.

DQM Data I/O Mask Function.

The SDQM[1:0] pins provide a byte-masking capability on 8-bit writes to SDRAM.

DRAM (Dynamic Random Access Memory).

A type of semiconductor memory in which the data is stored as electrical charges in an array of cells, each consisting of a capacitor and a transistor. The cells are arranged on a chip in a grid of rows and columns. Since the capacitors discharge gradually—and the cells lose their information—the array of cells has to be refreshed periodically.

DSP.

Digital signal processor

EAB.

External access bus

EBC.

External bus controller

EBIU.

External bus interface unit

edge-sensitive interrupt.

A signal or interrupt the processor detects if the input signal is high (inactive) on one cycle and low (active) on the next cycle when sampled on the rising edge of CLKIN.

Endian format.

The ordering of bytes in a multibyte number.

EPB.

External port bus

EPROM (Erasable Programmable Read-Only Memory).

A type of semiconductor memory in which the data is stored as electrical charges in isolated ("floating") transistor gates that retain their charges almost indefinitely without an external power supply. An EPROM is programmed by "injecting" charge into the floating gates—a process that requires relatively high voltage (usually 12V – 25V). Ultraviolet light, applied to the chip's surface through a quartz window in the package, discharges the floating gates, allowing the chip to be reprogrammed.

EVT (Event Vector Table).

A table stored in memory that contains sixteen 32-bit entries; each entry contains a vector address for an interrupt service routine (ISR). When an event occurs, instruction fetch starts at the address location in the corresponding EVT entry. See *ISR*.

exclusive, clean.

The state of a data cache line indicating the line is valid and the data contained in the line matches that in the source memory. The data in a clean cache line does not need to be written to source memory before it is replaced.

External Access Bus (EAB).

A bus mastered by the core memory management unit to access external memory.

External Bus Controller (EBC).

A component that provides arbitration between the External Access Bus (EAB) and the DMA External Bus (DEB), granting at most one requester per cycle.

External Bus Interface Unit (EBIU).

A component that provides glueless interfaces to external memories. It services requests for external memory from the core or from a DMA channel.

external port.

A channel or port that extends the processor's internal address and data buses off-chip, providing the processor's interface to off-chip memory and peripherals.

External Port Bus (EPB).

A bus that connects the output of the EBIU to external devices.

FFT (Fast Fourier Transform).

An algorithm for computing the Fourier transform of a set of discrete data values. The FFT expresses a finite set of data points, for example a periodic sampling of a real-world signal, in terms of its component frequencies. Or conversely, the FFT reconstructs a signal from the frequency data. The FFT can also be used to multiply two polynomials.

FIFO (First In, First Out).

A hardware buffer or data structure from which items are taken out in the same order they were put in.

flash memory.

A type of single transistor cell, erasable memory in which erasing can only be done in blocks or for the entire chip.

fully associative.

Cache architecture where each line can be placed anywhere in the cache.

glueless.

No external hardware is required.

Harvard architecture.

A processor memory architecture that uses separate buses for program and data storage. The two buses let the processor fetch a data word and an instruction word simultaneously.

HLL (High Level Language).

A programming language that provides some level of abstraction above assembly language, often using English-like statements, where each command or statement corresponds to several machine instructions.

I^2C.

A bus standard specified in the *Philips I^2C Bus Specification version 2.1* dated January 2000.

IDLE.

An instruction that causes the processor to cease operations, holding its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

index.

Address portion that is used to select an array element (for example, line index).

Index registers.

A Data Address Generator (DAG) register that holds an address and acts as a pointer to memory.

indirect branches.

Jump or call/return instructions that use a dynamic address from the data address generator, evaluated at runtime.

input clock.

Device that generates a steady stream of timing signals to provide the frequency, duty cycle, and stability to allow accurate internal clock multiplication via the phase locked loop (PLL) module.

internal memory bank.

There are up to 4 internal memory banks on a given SDRAM. Each of these banks can be accessed with the bank select lines `BA[1:0]`. The bank address can be thought of as part of the row address.

interrupt.

An event that suspends normal processing and temporarily diverts the flow of control through an interrupt service routine (ISR). See *ISR*.

invalid.

Describes the state of a cache line. When a cache line is invalid, a cache line match cannot occur.

IrDA (Infrared Data Association).

A nonprofit trade association that established standards for ensuring the quality and interoperability of devices using the infrared spectrum.

isochronous.

Processes where data must be delivered within certain time constraints.

ISR (Interrupt Service Routine).

Software that is executed when a specific interrupt occurs. A table stored in low memory contains pointers, also called vectors, that direct the processor to the corresponding ISR. See *EVT*.

JTAG (Joint Test Action Group).

An IEEE Standards working group that defines the IEEE 1149.1 standard for a test access port for testing electronic devices.

JTAG port.

A channel or port that supports the IEEE standard 1149.1 JTAG standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system.

jump.

A permanent transfer of the program flow to another part of program memory.

latency.

The overhead time used to find the correct place for memory access and preparing to access it.

Least Recently Used algorithm.

Replacement algorithm used by cache that first replaces lines that have been unused for the longest time.

Least Significant Bit (LSB).

The last or rightmost bit in the normal representation of a binary number—the bit of a binary number giving the number of ones.

Length registers.

A Data Address Generator (DAG) register that specifies the range of addresses in a circular buffer.

Level 1 (L1) memory.

Memory that is directly accessed by the core with no intervening memory subsystems between it and the core.

Level 2 (L2) memory.

Memory that is at least one level removed from the core. L2 memory has a larger capacity than L1 memory, but it requires additional latency to access.

level-sensitive interrupts.

A signal or interrupt that the processor detects if the input signal is low (active) when sampled on the rising edge of `CLKIN`.

LIFO (Last In, First Out).

A data structure from which the next item taken out is the most recent item put in.

little endian.

The native data store format of the processor. Words and half words are stored in memory (and registers) with the least significant byte at the lowest byte address and the most significant byte at the highest byte address of the data storage location.

loop.

A sequence of instructions that executes several times.

LRU.

See *Least Recently Used algorithm*

LSB.

See *Least Significant Bit*

MAC (Media Access Control).

The Ethernet MAC provides a 10/100M bit/s Ethernet interface, compliant to IEEE Std. 802.3-2002, between an MII (Media Independent Interface) and the Blackfin peripheral subsystem.

MAC (Multiply/Accumulate).

A mathematical operation that multiplies two numbers and then adds a third to get the result (see *Multiply Accumulator*).

Memory Management Unit (MMU).

A component of the processor that supports protection and selective caching of memory by using Cacheability Protection Lookaside Buffers (CPLBs).

Mode register.

Internal configuration registers within SDRAM devices which allow specification of the SDRAM device's functionality.

modified addressing.

The process whereby the Data Address Generator (DAG) produces an address that is incremented by a value or the contents of a register.

Modify register.

A Data Address Generator (DAG) register that provides the increment or step size by which an index register is pre- or post-modified during a register move.

MMR (Memory-mapped Register).

A specific location in main memory used by the processor as if it were a register.

MMU.

See *Memory Management Unit*

MSB (Most Significant Bit).

The first or leftmost bit in the normal representation of a binary number—the bit of a binary number with the greatest weight (2(n-1)).

multifunction computations.

The parallel execution of multiple computational instructions. These instructions complete in a single cycle, and they combine parallel operation of the computational units and memory accesses. The multiple operations perform the same as if they were in corresponding single function computations.

multiplier.

A computational unit that performs fixed-point multiplication and executes fixed-point multiply/add and multiply/subtract operations.

NMI (Nonmaskable Interrupt).

A high priority interrupt that cannot be disabled by another interrupt.

NRZ (Non-return-to-Zero).

A binary encoding scheme in which a 1 is represented by a change in the signal and a 0 by no change—there is no return to a reference (0) voltage between encoded bits. This method eliminates the need for a clock signal.

NRZI (Non-return-to-Zero Inverted).

A binary encoding scheme in which a 0 is represented by a change in the signal and a 1 is represented by no change—there is no return to a reference (0) voltage between encoded bits. This method eliminates the need for a clock signal.

orthogonal.

The characteristic of being independent. An orthogonal instruction set allows any register to be used in an instruction that references a register.

PAB.

See *Peripheral Access Bus*

page size.

The amount of memory which has the same row address and can be accessed with successive read or write commands without needing to activate another row.

Parallel Peripheral Interface (PPI).

The PPI is a half-duplex, bidirectional port accommodating up to 16 bits of data. It has a dedicated clock pin and three multiplexed frame sync pins.

PC (Program Counter).

A register that contains the address of the next instruction to be executed.

peripheral.

Functional blocks not included as part of the core, and typically used to support system level operations.

Peripheral Access Bus (PAB).

A bus used to provide access to EBIU memory-mapped registers.

PF (Programmable Flag).

General-purpose I/O pins. Each PF pin can be individually configured as either an input or an output pin, and each PF pin can be further configured to generate an interrupt.

Phase Locked Loop (PLL).

An on-chip frequency synthesizer that produces a full speed master clock from a lower frequency input clock signal.

PLL.

See *Phase Locked Loop*

PPI.

See *Parallel Peripheral Interface*

precision.

The number of bits after the binary point in the storage format for the number.

post-modify addressing.

The process in which the Data Address Generator (DAG) provides an address during a data move and auto-increments after the instruction is executed.

precharge command.

The precharge command closes a specific active page in an internal bank and the precharge all command closes all 4 active pages in all 4 banks.

pre-modify addressing.

The process in which the Data Address Generator (DAG) provides an address during a data move and auto-increments before the instruction is executed.

PWM (Pulse Width Modulation).

Also called Pulse Duration Modulation (PDM), PWM is a pulse modulation technique in which the duration of the pulses is varied by the modulating voltage.

RAS (Row Address Strobe).

A signal sent from the SDC to a DRAM device to indicate validity of row address lines.

Real-Time Clock (RTC).

A component that generates timing pulses for the digital watch features of the processor, including time of day, alarm, and stopwatch countdown features.

ROM (Read-Only Memory).

A data storage device manufactured with fixed contents. This term is most often used to refer to non-volatile semiconductor memory.

RTC.

See *Real-Time Clock*

RZ (Return-to-Zero modulation).

A binary encoding scheme in which two signal pulses are used for every bit. A 0 is represented by a change from the low voltage level to the high voltage level; a 1 is represented by a change from the high voltage level to the low voltage level. A return to a reference (0) voltage is made between encoded bits.

RZI (Return-to-Zero-Inverted modulation).

A binary encoding scheme in which two signal pulses are used for every bit. A 1 is represented by a change from the low voltage level to the high voltage level; a 0 is represented by a change from the high voltage level to

the low voltage level. A return to a reference (0) voltage is made between encoded bits.

saturation (ALU saturation mode).

A state in which all positive fixed-point overflows return the maximum positive fixed-point number, and all negative overflows return the maximum negative number.

SDC (SDRAM Controller).

A configurable memory controller supporting a bank of synchronous memory consisting of SDRAM.

SDRAM (Synchronous Dynamic Random Access Memory).

A form of DRAM that includes a clock signal with its other control signals. This clock signal allows SDRAM devices to support "burst" access modes that clock out a series of successive bits.

SDRAM bank.

Region of external memory that can be configured to be 16M bytes, 32M bytes, 64M bytes, or 128M bytes and is selected by the $\overline{\texttt{SMS}}$ pin.

Self-Refresh.

When the SDRAM is in self-refresh mode, the SDRAM's internal timer initiates auto-refresh cycles periodically, without external control input. The SDRAM Controller (SDC) must issue a series of commands including the self-refresh command to put SDRAM into low power mode, and it must issue another series of commands to exit self-refresh mode. Entering self-refresh mode is programmed in the SDRAM memory global control register (`EBIU_SDGCTL`) and any access to the SDRAM address space causes the SDC to exit SDRAM from self-refresh mode. See "Enter Self-Refresh Mode" on page 7-34 and "Exit Self-Refresh Mode" on page 7-34.

Serial Peripheral Interface (SPI).

A synchronous serial protocol used to connect integrated circuits.

serial ports (SPORTs).

A high speed synchronous input/output device on the processor. The processor uses two synchronous serial ports that provide inexpensive interfaces to a wide variety of digital and mixed-signal peripheral devices.

set.

A group of *N*-line storage locations in the ways of an *N*-way cache, selected by the index field of the address.

set associative.

Cache architecture that limits line placement to a number of sets (or ways).

shifter.

A computational unit that completes logical and arithmetic shifts.

SIC (System Interrupt Controller).

Part of the processor's two-level event control mechanism. The SIC works with the Core Event Controller (CEC) to prioritize and control all system interrupts. The SIC provides mapping between the peripheral interrupt sources and the prioritized general-purpose interrupt inputs of the core.

SIMD (Single Instruction, Multiple Data).

A parallel computer architecture in which multiple data operands are processed simultaneously using one instruction.

SP (Stack Pointer).

A register that points to the top of the stack.

SPI.

See *Serial Peripheral Interface*

SRAM.

See *Static Random Access Memory*

stack.

A data structure for storing items that are to be accessed in Last In, First Out (LIFO) order. When a data item is added to the stack, it is "pushed"; when a data item is removed from the stack, it is "popped."

Static Random Access Memory (SRAM).

Very fast read/write memory that does not require periodic refreshing.

system.

The system includes the peripheral set (timers, real-time clock, programmable flags, UART, SPORTs, PPI, and SPIs), the external memory controller (EBIU), the memory DMA controller, as well as the interfaces between these peripherals, and the optional, external (off-chip) resources.

System clock (SCLK).

A component that delivers clock pulses at a frequency determined by a programmable divider ratio within the PLL.

System Interrupt Controller (SIC).

Component that maps and routes events from peripheral interrupt sources to the prioritized, general-purpose interrupt inputs of the Core Event Controller (CEC).

TAP (Test Access Port).

See *JTAG port*

TDM.

See *Time Division Multiplexing*

Time Division Multiplexing (TDM).

A method used for transmitting separate signals over a single channel. Transmission time is broken into segments, each of which carries one element. Each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of the 24 channels.

TWI.

See *Two Wire Interface*

Two Wire Interface (TWI).

The TWI controller allows a device to interface to an Inter IC bus as specified by the *Philips I^2C Bus Specification version 2.1* dated January 2000. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the `SCL` rate, to and from other TWI devices.

UART.

See *Universal Asynchronous Receiver Transmitter*

Universal Asynchronous Receiver Transmitter (UART).

A module that contains both the receiving and transmitting circuits required for asynchronous serial communication.

Valid.

A state bit (stored along with the tag) that indicates the corresponding tag and data are current and correct and can be used to satisfy memory access requests.

victim.

A dirty cache line that must be written to memory before it can be replaced to free space for a cache line allocation.

Von Neumann architecture.

The architecture used by most non-DSP microprocessors. This architecture uses a single address and data bus for memory access.

Way.

An array of line storage elements in an *N*-Way cache.

W1C.

See *Write-1-to-Clear*

W1S.

See *Write-1-to-Set*

Write-1-to-Clear (W1C) bit.

A control or status bit that can be cleared (= 0) by being written to with 1.

Write-1-to-Set (W1S) bit.

A control or status bit that is set by writing 1 to it. It cannot be cleared by writing 0 to it.

write back.

A cache write policy (also known as copyback). The write data is written only to the cache line. The modified cache line is written to source memory only when it is replaced.

write through.

A cache write policy (also known as store through). The write data is written to both the cache line and to source memory. The modified cache line is *not* written to the source memory when it is replaced.

I INDEX

Symbols

'A' or 'B' device indicator (B_DEVICE) bit, 26-132, 26-134
'B' or 'A' device indicator (B_DEVICE) bit, 26-132, 26-134
µ-law companding, 24-24, 24-29

Numerics

BCODE, 16-47, 16-48
OTP_EBIU_SDGCTL, 17-114
16-bit flash interface, 19-5
16-bit SRAM interface, 19-5
24 hours event flag bit, 14-21
24 hours interrupt enable bit, 14-20
2D DMA, 6-12
2X input clock, 24-25
BCODE, 17-102
BMODE, 16-47, 16-48, 17-102
OTP_EBIU_SDGCTL, 17-114
OTP_EBIU_SDBCTL, 17-114
5 volt tolerance, 19-11
8/16 bit mode (HOSTDP_DATA_SIZE), 8-26
8-bit flash interface, 19-5
8-bit SRAM interface, 19-5

A

A[10] pin, 7-54
$\overline{\text{ABE}}$[1:0] pins, 7-17
aborted frames, MAC, 21-16
aborts, DMA, 6-29
accesses
  off-core, 2-5
  to internal memory, 3-1
access way/instruction address bit 11 bit, 3-7
active descriptor queue, and DMA synchronization, 6-61
active low/high frame syncs, serial port, 24-34
active mode, 1-26, 18-9
ACTIVE_PLLDISABLED bit, 18-28
ACTIVE_PLLENABLED bit, 18-28
active video only mode, PPI, 15-10
ADCs, connecting to, 24-2
address filter evaluation, MAC, 21-13
address mapping, SDRAM, 7-26
alarm clock, RTC, 14-2
alarm event flag bit, 14-21
alarm interrupt enable bit, 14-20
A-law companding, 24-24, 24-29
AlignmentErrors register, 21-54
alternate frame sync mode, 24-37
alternate timing, serial port, 24-36
AMBEN[2:0] field, 7-18, 7-20
AMC, 1-9, 1-13
  bus contention, 7-10
  EBIU block diagram, 7-4
  features, 7-8
  signals, 7-9
  timing parameters, 7-18
AMCKEN bit, 7-18, 7-20
AMS, 7-8
ANAK (address not acknowledged) bit, 23-35, 23-37

application data, loading, 17-1
arbitration
  DAB, 2-8, 2-9
  DCB, 2-8, 2-9
  DEB, 2-8, 2-9
  EAB, 2-15
  latency, 2-14
  TWI, 23-8
architecture, memory, 3-1
ARDY pin, 7-11, 7-15
array access bit, 3-7
ASIC/FPGA designs, 7-1
ASTP (enable automatic pad stripping) bit, 21-17, 21-61, 21-67
asynchronous
  interfaces, supported, 7-1
  memory, 3-1, 7-2, 7-8
  memory bank address range (table), 7-8
  memory controller, 1-13
  read, 7-11
  serial communications, 25-5
  write, 7-13
asynchronous memory bank control 0 register (EBIU_AMBCTL0), 7-21
asynchronous memory bank control 1 register (EBIU_AMBCTL1), 7-22
asynchronous memory bank control (EBIU_AMBCTLx), 7-20
asynchronous memory controller, *See* AMC
asynchronous memory global control (EBIU_AMGCTL), 7-17
ASYNC memory banks, 7-3
atomic operations, 19-3
autobaud, and general-purpose timers, 10-31
autobaud detection, 25-14
autobuffer mode, 6-12, 6-29, 6-70
AUTOCLEAR_R (RxPktRdy autoclear enable) bit, 26-121
auto-refresh
  command, 7-34, 7-51
  timing, 7-59
AUTOREQ_RH (autoset ReqPkt) bit, 26-121
autoset ReqPkt (AUTOREQ_R) bit, 26-121
AUTOSET_T (TxPktRdy autoset enable) bit, 26-115
avoiding bus contention, 7-10

B

B0HT[1:0] field, 7-21
B0RAT[3:0] field, 7-21
B0RDYEN bit, 7-21
B0RDYPOL bit, 7-21
B0ST[1:0] field, 7-21
B0TT[1:0] field, 7-21
B0WAT[3:0] field, 7-21
B1HT[1:0] field, 7-21
B1RAT[3:0] field, 7-21
B1RDYEN bit, 7-21
B1RDYPOL bit, 7-21
B1ST[1:0] field, 7-21
B1TT[1:0] field, 7-21
B1WAT[3:0] field, 7-21
B2HT[1:0] field, 7-22
B2RAT[3:0] field, 7-22
B2RDYEN bit, 7-22
B2RDYPOL bit, 7-22
B2ST[1:0] field, 7-22
B2TT[1:0] field, 7-22
B2WAT[3:0] field, 7-22
B3HT[1:0] field, 7-22
B3RAT[3:0] field, 7-22
B3RDYEN bit, 7-22
B3RDYPOL bit, 7-22
B3ST[1:0] field, 7-22
B3TT[1:0] field, 7-22
B3WAT[3:0] field, 7-22

BA[1:0] pins, 7-32
bable or reset indicator (RESET_OR_BABLE_B) bit, 26-107
bable or reset IRQ enable (RESET_OR_BABLE_BE) bit, 26-108
bandwidth, and memory DMA operations, 6-48
bank
  address, 7-63
  size, 7-25, 7-63
  size encodings (table), 7-27
  width, 7-25
bank activate command, 7-33, G-1
baud rate
  SPI, 22-35
  UART, 25-7, 25-14
baud rate[15:0] field, 22-35
BCINIT[15:0] field, 6-87
BCOUNT[15:0] field, 6-88
B_DEVICE ('A' or 'B' device indicator) bit, 26-132, 26-134
BDI (block done interrupt generated) bit, 6-86
BDIE (block done interrupt enable) bit, 6-41, 6-86
BDR (burst DMA requests) bit, 8-26, 8-29
BFLAG_AUX, 17-121
BFLAG_CALLBACK, 17-121
BFLAG_FASTREAD, 17-120
BFLAG_FILL, 17-121
BFLAG_FINAL, 17-121
BFLAG_FIRST, 17-121
BFLAG_HDRINDIRECT, 17-120
BFLAG_HOOK, 17-120
BFLAG_IGNORE, 17-121
BFLAG_INDIRECT, 17-121
BFLAG_INIT, 17-121
BFLAG_NEXTDXE, 17-120
BFLAG_NOAUTO, 17-120
BFLAG_NONRESTORE, 17-120
BFLAG_PERIPHERAL, 17-120
BFLAG_QUICKBOOT, 17-121
BFLAG_RESET, 17-120
BFLAG_RETURN, 17-120
BFLAG_SAVE, 17-121
BFLAG_SLAVE, 17-120
BFLAG_TYPE, 17-120
BFLAG_WAKEUP, 17-120
BI (break interrupt) bit, 25-25, 25-26
binary decode, B-4
bin x bit, 21-71, 21-72
bit 15 overflow interrupt enable (COV15IE) bit, 13-21
bit 15 overflow interrupt identifier (COV15II) bit, 13-22
bit 31 overflow interrupt enable (COV31IE) bit, 13-21
bit 31 overflow interrupt identifier (COV31II) bit, 13-22
bit order, selecting, 24-28
bit rate generation, 25-13
BK_DATECODE (Boot Code Date Code), 17-104
BK_DAY, 17-104
BK_ID, 17-103
BK_MONTH, 17-104
BK_ONES, 17-106
BK_PROJECT, 17-103
BKPRSEN (enable backpressure) bit, 21-75
BK_REVISION (Boot Code Revision Control), 17-103
BK_UPDATE, 17-103
BK_VERSION, 17-103
BK_YEAR, 17-104
BK_ZEROS, 17-105
Blackfin processor family
  I/O memory space, 1-9
  memory architecture, 1-7

block, DMA, 6-9
block code field, 17-25
block count, DMA, 6-38
block diagrams
  bus hierarchy, 2-3
  core, 2-5
  core timer, 11-2
  DMA controller, 6-105
  EBIU, 7-4
  general-purpose timers, 10-58
  HOSTDP, 8-4
  MAC, 21-3
  PLL, 18-3
  PPI, 15-3
  processor, 1-7
  RTC, 14-4
  SDRAM, 7-56, 7-57
  SPI, 22-4
  SPORT, 24-6
  TWI, 23-3
  UART, 25-3
  watchdog timer, 12-3
block done interrupt, DMA, 6-41
Block Flags, 17-27
block transfers, DMA, 6-38
BMODE00_DIS, 17-113
BMODE01_DIS, 17-113
BMODE02_DIS, 17-113
BMODE03_DIS, 17-113
BMODE04_DIS, 17-113
BMODE05_DIS, 17-113
BMODE06_DIS, 17-113
BMODE07_DIS, 17-113
BMODE08_DIS, 17-113
BMODE09_DIS, 17-113
BMODE10_DIS, 17-113
BMODE11_DIS, 17-113
BMODE12_DIS, 17-113
BMODE13_DIS, 17-113
BMODE14_DIS, 17-113
BMODE15_DIS, 17-113
BMODE[2:0] pins, 17-6
BMODE pins, 17-1
BNDMODE (boundary register mode)
    bits, 13-20
BOLMT[1:0] field, 21-61, 21-63
boost PLL amplitude (TM_PLL_VCO)
    bit, 26-141
boot
  call boot kernel at run time, 17-48
  load function, 17-47
  manager, 17-52
  quick, 17-41
  ROM functions, 17-53
  streams
    multi-DXE, 17-53
Boot Code Date Code
    (BK_DATECODE), 17-104
Boot Code Revision Control
    (BK_REVISION), 17-103
boot host wait
  HWAIT, 17-31
booting
  BFROM_MEMBOOT, 17-52
  BFROM_NANDBOOT, 17-52
  BFROM_OTPBOOT, 17-52
  BFROM_SPIBOOT, 17-52
  BFROM_TWIBOOT, 17-52
  boot stream, 17-21
  host boot scenarios, 17-22
  host DMA boot modes, 17-81
  indirect, 17-42
  memory locations, 17-22
  NAND flash boot mode, 17-86
  SPI slave mode, 17-69
  TWI master mode, 17-72
  TWI slave mode, 17-75
booting modes, 17-2
boot kernel, 17-1
Boot Management, 17-51

boot mode
FIFO boot, 17-63
flash boot, 17-59
no-boot, 17-59
SDRAM boot, 17-63
SPI device detection, 17-67
boot ROM
internal, 17-1
memory space, 3-5
boot stream, 17-1, 17-21
boot termination, 17-33
boundary register mode (BNDMODE) bits, 13-20
boundary-scan architecture, B-2
boundary-scan register, B-7
BroadcastFramesReceivedOK register, 21-55
BroadcastFramesXmittedOK register, 21-59
broadcast mode, 22-9, 22-15, 22-16
BT_EN (bus timeout enable) bit, 8-26
buffer registers, timers, 10-44
BUFRDERR (buffer read error) bit, 23-35, 23-37
BUFWRERR (buffer write error) bit, 23-35, 23-37
burst
length, 7-32, G-2
type, 7-32, G-2
burst DMA requests (BDR) bit, 8-26, 8-29
BURST_MODE (DMA burst mode selection) bits, 26-144
bus agents
DAB, 2-13
PAB, 2-6
BUSBUSY (bus busy) bit, 23-35, 23-36
bus contention, avoiding, 7-10, 19-7
bus cycles
asynchronous read, 7-12
asynchronous write, 7-13
bus error, EBIU, 7-7
BUSERROR (DMA bus error) bit, 26-144
buses
See also DAB, DCB, DEB, EAB, EPB, PAB
bandwidth, 1-6
core, 2-4
hierarchy, 2-3
on-chip, 2-1
PAB, 2-6
peripheral, 2-6
and peripherals, 1-6
prioritization and DMA, 6-49
bus standard, I^2C, 1-15
bus timeout enable (BT_EN) bit, 8-26
bypass
capacitor placement, 19-10
BYPASS bit, 18-26, 18-27
BYPASS instruction, B-6
BYPASS register, B-6
byte
address, 7-63
enables, 7-17
byte enable x bit, 21-82, 21-83, 21-84, 21-85

C

callback routines, 17-43
capacitors, 19-9
capture mode, *See* WDTH_CAP mode
CAPWKFRM bit, 21-79, 21-80
CarrierSenseErrors register, 21-58
CAS latency, 7-33, 7-35
CAW, 7-63
CCIR-656, *See* ITU-R 656
CCITT G.711 specification, 24-29
CCLK (core clock), 18-5
status by operating mode, 18-8
CCLK (core processor clock), 2-3
CCOR bit, 21-43, 21-121

CDGINV (CDG pin polarity invert) bit, 13-20
CDG pin polarity invert (CDGINV) bit, 13-20
CDPRIO bit, 7-18, 7-20, 7-43
channels
  defined, serial, 24-23
  serial port TDM, 24-23
  serial select offset, 24-23
charge VBUS end interrupt enable (CHRG_VBUS_END_ENA) bit, 26-136
charge VBUS start interrupt enable (CHRG_VBUS_START_ENA) bit, 26-136
CHNL[9:0] field, 24-68, 24-69
CHRG_VBUS_END_ENA (charge VBUS end interrupt enable) bit, 26-136
CHRG_VBUS_START_ENA (charge VBUS start interrupt enable) bit, 26-136
circuit board testing, B-1, B-6
circular addressing, 6-58
CKELOW (CKE low reset) bit, 18-29, 18-30
CL, 7-35
CL[1:0] field, 7-66, 7-67
CLEAR_DATATOGGLE_R (reset endpoint data toggle) bit, 26-121
CLEAR_DATATOGGLE_T (reset endpoint data toggle) bit, 26-115
clearing interrupt requests, 5-13
clear Pxn bit, 9-33
clear Pxn interrupt A enable bit, 9-39
clear Pxn interrupt B enable bit, 9-40
CLKBUFOE (CLKIN buffer output enable) bit, 18-28, 18-29, 18-30
CLKBUF pin, 21-4, 21-47
CLKDIV (clock divisor) bits, 26-142
CLKHI[7:0] field, 23-28
CLKIN (input clock), 1-25, 2-3, 18-1, 18-2
CLKLOW[7:0] field, 23-28
CLKOUT pin, 7-7, 7-67
  disabling, 7-72
CLK_SEL (timer clock select) bit, 10-12, 10-20, 10-41, 10-46
clock
  clock signals, 1-25
  control, 18-1
  EBIU, 7-1
  external, 1-25
  frequency for SPORT, 24-65
  internal, 2-3
  MAC, 21-4
  managing, 19-1
  peripheral, 18-7
  RTC, 14-5
  source for general-purpose timers, 10-3
  SPI clock signal, 22-4
  system, 1-25
  system (SCLK), 19-2
  types, 19-1
clock divide modulus registers, 24-65
clock divisor (CLKDIV) bits, 26-142
clock domain synchronization, PPI, 15-15
clock input (CLKIN) pin, 19-1
clock phase, SPI, 22-12, 22-14
clock polarity, SPI, 22-12
clock rate
  core timer, 11-1
  SPORT, 24-2
clock ratio, changing, 18-6
clocks, overview, 1-25
clock signals, 1-25
CNFG_PEND (config pending) bit, 8-29
CNOS (tuning of DPHY clocks) bits, 26-140

CNT_COMMAND (command) register, 13-19, 13-22
CNT_CONFIG (configuration) register, 13-19, 13-20
CNT_COUNTER (counter) register, 13-19, 13-26
CNT_DEBOUNCE (debounce) register, 13-19, 13-25
CNTE (counter enable) bit, 13-20
CNT_IMASK (interrupt mask) register, 13-19, 13-21
CNT_MAX (maximal count) register, 13-19, 13-26
CNT_MIN (minimal count) register, 13-19, 13-26
CNTMODE (counter operating mode) bits, 13-20
CNT_STATUS (status) register, 13-19, 13-22
codecs, connecting to, 24-2
column address, 7-63
  strobe latency, 7-33, G-4
column read/write, SDRAM, 7-31
command (CNT_COMMAND) register, 13-19, 13-22
command inhibit command, 7-53
commands
  auto-refresh, 7-34, 7-51, 7-59
  bank activate, 7-33, G-1
  command inhibit, 7-53
  DMA control, 6-32, 6-33
  EMRS, 7-48
  ERMS, 7-33
  MRS, 7-33, 7-47
  no operation (NOP), 7-53
  precharge, 7-34, G-18
  precharge all, 7-34, 7-51
  read, 7-34
  read/write, 7-49
  SDC, 7-46
  self-refresh, 7-52
  transfer initiate, 22-18, 22-19
  write, 7-34
  write with data mask, 7-50
companding, 24-16, 24-24
  defined, 24-29
  lengths supported, 24-30
  multichannel operations, 24-24
COMPLETE (DMA complete) bit, 8-29
config pending (CNFG_PEND) bit, 8-29
configuration
  SDC, 7-54
  SDRAM, 7-24
  SPORT, 24-11
configuration (CNT_CONFIG) register, 13-19, 13-20
congestion, on DMA channels, 6-46
CONN_B (connection indicator) bit, 26-107
CONN_BE (connection IRQ enable) bit, 26-108
connection indicator (CONN_B) bit, 26-107
connection IRQ enable (CONN_BE) bit, 26-108
contention, bus, avoiding, 7-10
continuous polling, and MMC register, 21-44
continuous transition, DMA, 6-27
control bit summary, general-purpose timers, 10-46
control byte sequences, PPI, 15-8
control frames, MAC, 21-16
control register
  data memory, 3-6
  EBIU, 7-6

core
block diagram, 2-5
core bus, 2-4
core clock (CCLK), 18-5, 19-2
core clock/system clock ratio control, 18-5
timer, 5-5
waking from idle state, 5-6
core and system reset, code example, 17-138
core clock, *See* CCLK
core clock (CCLK), 11-2
core double-fault reset, 17-6
core event controller (CEC), 5-2
core-only software reset, 17-6
core select (CSEL) bits, 18-26
core timer, 11-1 to 11-8
block diagram, 11-2
clock rate, 11-1
features, 11-2
initialization, 11-3
internal interfaces, 11-3
low power mode, 11-3
operation, 11-3
registers, 11-4
scaling, 11-7
core timer control (TCNTL) register, 11-3, 11-5
core timer count (TCOUNT) register, 11-3, 11-5
core timer scale (TSCALE) register, 11-3, 11-7
counter, RTC, 14-2
counter (CNT_COUNTER) register, 13-19, 13-26
counter enable (CNTE) bit, 13-20
counter operating mode (CNTMODE) bits, 13-20
COUNT_TIMEOUT (host timeout count) bits, 8-31
count to zero interrupt enable (CZEROIE) bit, 13-21
count to zero interrupt identifier (CZEROII) bit, 13-22
count value[15:0] field, 11-6
count value[31:16] field, 11-6
COV15IE (bit 15 overflow interrupt enable) bit, 13-21
COV15II (bit 15 overflow interrupt identifier) bit, 13-22
COV31IE (bit 31 overflow interrupt enable) bit, 13-21
COV31II (bit 31 overflow interrupt identifier) bit, 13-22
CPHA bit, 22-37
CPOL bit, 22-37
CRC-16 hash value calculation, 21-38
CRC-32 calculation, MAC, 21-70
CRC32 checksum generation, 17-46
CRC state, MAC, 21-36
CROLL bit, 21-44, 21-121, 21-122
CROSSCORE software, 1-29
crosstalk, 19-9
crystal
external, 1-25
CSEL[1:0] field, 18-5, 18-26, 19-2
CSR_HBR (USB hibernate signal) bit, 26-140
CSR_RSTD (USB pu/pd restore control) bit, 26-140
CTYPE (DMA channel type) bit, 6-68
CUD and CDZ input disable (INPDIS) bit, 13-20
CUDINV (CUD pin polarity invert) bit, 13-20
CUD pin polarity invert (CUDINV) bit, 13-20
current address field, 6-77

current address registers
(DMAx_CURR_ADDR), 6-76
(MDMA_yy_CURR_ADDR), 6-76
current descriptor pointer
(DMAx_CURR_DESC_PTR)
registers, 6-83
current descriptor pointer
(MDMA_yy_CURR_DESC_PTR)
registers, 6-83
current inner loop count registers
(DMAx_CURR_X_COUNT), 6-78, 6-79
(MDMA_yy_CURR_X_COUNT), 6-78, 6-79
current outer loop count registers
(DMAx_CURR_Y_COUNT), 6-81
(MDMA_yy_CURR_Y_COUNT), 6-81
current time, 14-13
CURR_X_COUNT[15:0] field, 6-79
CURR_Y_COUNT[15:0] field, 6-81
customer support, -lx
CZEROIE (count to zero interrupt enable) bit, 13-21
CZEROII (count to zero interrupt identifier) bit, 13-22
CZMEIE (CZM error interrupt enable) bit, 13-21
CZMEII (CZM error interrupt identifier) bit, 13-22
CZM error interrupt enable (CZMEIE) bit, 13-21
CZM error interrupt identifier (CZMEII) bit, 13-22
CZMIE (CZM pin interrupt enable) bit, 13-21
CZMII (CZM pin interrupt identifier) bit, 13-22
CZMINV (CZM pin polarity invert) bit, 13-20
CZM pin interrupt enable (CZMIE) bit, 13-21
CZM pin interrupt identifier (CZMII) bit, 13-22
CZM pin polarity invert (CZMINV) bit, 13-20
CZM zeroes counter enable (ZMZC) bit, 13-20
CZM zeroes counter interrupt enable (CZMZIE) bit, 13-21
CZM zeroes counter interrupt identifier (CZMZII) bit, 13-22
CZMZIE (CZM zeroes counter interrupt enable) bit, 13-21
CZMZII (CZM zeroes counter interrupt identifier) bit, 13-22

D

DAB, 2-8, 6-5, 6-43, 6-92
arbitration, 2-8, 2-9
bus agents (masters), 2-13
clocking, 18-2
latencies, 2-14
performance, 2-14
throughput, 2-14
DAB_TRAFFIC_COUNT[2:0] field, 6-92
data
I/O mask function, 7-33
masks, 7-40
sampling, serial, 24-34
data bank access bit, 3-7
data cache select/address bit 14 bit, 3-7
data corruption, avoiding with SPI, 22-15
data-driven interrupts, 6-75
DATAEND (data end indicator) bit, 26-111
data end indicator (DATAEND) bit, 26-111

DATAERROR_R (load error indicator) bit, 26-121
data formats, SPORT, 24-29
data input modes for PPI, 15-14 to 15-17
data/instruction access bit, 3-7
data interrupt timing select (DI_SEL) bit, 8-6
data memory control (DMEM_CONTROL) register, 3-6
data memory control register (DMEM_CONTROL), 3-6
data move, serial port operations, 24-39
data output modes for PPI, 15-17 to 15-19
data packet in FIFO indicator (FIFO_NOT_EMPTY_T) bit, 26-115
data packet in FIFO indicator (RXPKTRDY_R) bit, 26-121
data packet in FIFO indicator (TXPKTRDY) bit, 26-111
data packet in FIFO indicator (TXPKTRDY_T) bit, 26-115
data packet receive indicator (RXPKTRDY) bit, 26-111
DATA_SIZE bit, 8-26
data structures, 17-115
  boot_struct, 17-117
  buffer_struct, 17-116
  header_struct, 17-115
data test command register (DTEST_COMMAND), 3-6
data transfers
  DMA, 2-14, 6-2
  SPI, 22-15
data word, serial data formats, 24-58
day[14:0] field, 14-22
day alarm event flag bit, 14-21
day alarm interrupt enable bit, 14-20
day counter[14:0] field, 14-20
DBF bit, 21-61, 21-65
DCB, 2-8, 6-5, 6-43, 6-93
  arbitration, 2-8, 2-9
DCB1_PRIO, 17-102
DC bit, 21-61, 21-64
DCBS (L1 data cache bank select) bit, 3-6
DCB_TRAFFIC_COUNT field, 6-93
DCB_TRAFFIC_PERIOD field, 6-93
DCIE (down count interrupt enable) bit, 13-21
DCII (down count interrupt identifier) bit, 13-22
DCNT[7:0] field, 23-32, 23-33
DEB, 2-8, 6-5, 6-43, 6-93
  arbitration, 2-8, 2-9
  and EBIU, 7-5
  frequency, 2-15
  performance, 2-15
DEB1_PRIO, 17-102
DEBE (debounce enable) bit, 13-20
debounce (CNT_DEBOUNCE) register, 13-19, 13-25
debounce enable (DEBE) bit, 13-20
DEB_ROT_PRIO, 17-102
DEB_TRAFFIC_COUNT field, 6-92
DEB_TRAFFIC_PERIOD field, 6-92
debugging, 1-30
  test point access, 19-11
deep sleep, and RTC, 14-10
deep sleep mode, 1-27, 7-45, 18-10
delaycount (PPI_DELAY) register, 15-34
descriptor
  array mode, DMA, 6-16, 6-70
  chains, DMA, 6-27
  list mode, DMA, 6-15, 6-70, 6-71
  pairs, DMA, 21-12
descriptor-based DMA, 6-14
descriptor queue, 6-58
  management, 6-58
  synchronization, 6-58, 6-59

descriptor structures
alternative, 21-26
DMA, 6-57
MDMA, 6-63
destination channels, memory DMA, 6-7
detected FIFO not empty (FIFO_FULL_R) bit, 26-121
development tools, 1-29
DF bit, 18-3, 18-4, 18-26, 18-27
DF (divide CLKIN by 2) bit, 26-141
DFETCH bit, 6-15, 6-22, 6-74
dFlags Word, Bits 15–0, 17-121
dFlags Word, Bits 31–16, 17-120
DFRESET, 16-47, 16-48, 17-102
DI_EN bit, 6-14, 6-69, 6-71
direct code execution, 17-35
initial header, 17-34, 17-36
DIRECTION (DMA Tx or Rx selection) bit, 26-144
direction errors, MAC, 21-29
direct memory access, *See* DMA
disable nyet handshake (DISNYET) bit, 26-121
disabling
PLL, 18-13
discarded frames, MAC, 21-15
discharge VBUS end interrupt enable (DISCHRG_VBUS_END_ENA) bit, 26-136
discharge VBUS start interrupt enable (DISCHRG_VBUS_START_ENA) bit, 26-136
DISCHRG_VBUS_END_ENA (discharge VBUS end interrupt enable) bit, 26-136
DISCHRG_VBUS_START_ENA (discharge VBUS start interrupt enable) bit, 26-136
DISCON_B (disconnect/session end indicator) bit, 26-107
DISCON_BE (disconnect/session end IRQ enable) bit, 26-108
disconnect/session end indicator (DISCON_B) bit, 26-107
disconnect/session end IRQ enable (DISCON_BE) bit, 26-108
DI_SEL bit, 6-69, 6-71
DI_SEL (data interrupt timing select) bit, 8-6
DISNYET (disable nyet handshake) bit, 26-121
DITFS (data-independent transmit frame sync select) bit, 24-38, 24-49, 24-52, 24-64
divide CLKIN by 2 (DF) bit, 26-141
divisor latch high byte[15:8] field, 25-31
divisor latch low byte[7:0] field, 25-31
divisor reset, UART, 25-31
DLAB (divisor latch access) bit, 25-22, 25-27, 25-28
DLEN[2:0] field, 15-26, 15-27
DMA, 6-1 to 6-104
1-D interrupt-driven, 6-55
1-D unsynchronized FIFO, 6-56
2-D, polled, 6-55
2-D array, example, 6-94
2-D interrupt-driven, 6-55
autobuffer mode, 6-12, 6-29, 6-70
bandwidth, 6-46
block count, 6-38
block diagram, 6-105
block done interrupt, 6-41
block transfers, 6-9, 6-38
buffer size, multichannel SPORT, 24-24
buses, 2-8
channel registers, 6-67
channels, 6-43
channels and control schemes, 6-51
channel-specific register names, 6-66
congestion, 6-46

DMA, 6-1 to 6-104 *(continued)*
connecting asynchronous FIFO, 6-39
continuous transfers using autobuffering, 6-54
continuous transition, 6-27
control command restrictions, 6-35
control commands, 6-32, 6-33
controllers, 1-11
data transfers, 6-2
descriptor array, 6-23
descriptor array mode, 6-16, 6-70
descriptor-based, 6-14
descriptor-based, initializing, 6-97
descriptor-based vs. register-based transfers, 6-3
descriptor chains, 6-27
descriptor element offsets, 6-16
descriptor list mode, 6-15, 6-70, 6-71
descriptor lists, 6-23
descriptor queue, 6-58
descriptors, and MAC, 21-124
descriptors, recommended size, 6-17
descriptor structures, 6-57
direction, 6-72
DMA error interrupt, 6-75
double buffer scheme, 6-55
and EBIU, 6-4
errors, 6-29, 6-31
example connection, receive, 6-41
example connection, transmit, 6-40
external interfaces, 6-4
features, 6-2
finish control command, 6-34, 6-35
first data memory access, 6-22
flow chart, 6-19, 6-20
FLOW mode, 6-17
FLOW value, 6-21
for SPI transmit, 22-11
handshake DMA, 1-12
handshake operation, 6-37
header file to define descriptor structures example, 6-98
HMDMA1 block enable example, 6-103
HMDMA with delayed processing example, 6-104
initializing, 6-18
internal interfaces, 6-5
and L1 memory, 6-5
large model mode, 6-71
latency, 6-25
linked list, 21-124
MAC configuration, 21-123
memory conflict, 6-50
memory DMA, 1-12, 6-7
memory DMA streams, 6-7
memory DMA transfers, 6-5
memory read, 6-26
operation flow, 6-18
orphan access, 6-29
overflow interrupt, 6-42
overview, 1-11
performance considerations, 6-44
peripheral, 6-6, 6-106
peripheral channels priority, 6-6
peripheral interrupts, 5-6
peripheral priority and default mapping, 6-106
pipelining requests, 6-38
polling DMA status example, 6-96
polling registers, 6-52
and PPI, 15-37
prioritization and traffic control, 6-46 to 6-51
programming examples, 6-93 to 6-104
receive, 6-27
refresh, 6-23
register-based, 6-10
register-based 2D memory DMA example, 6-94
register naming conventions, 6-67

remapping peripheral assignment, 6-6
request data control command, 6-35
request data urgent control command, 6-35
restart control command, 6-33, 6-34
round robin operation, 6-49
serial port block transfers, 24-39
single-buffer transfers, 6-54
small model mode, 6-70
software management, 6-51
software-triggered descriptor fetch example, 6-100
and SPI, 22-11
SPI data transmission, 22-43
SPI master, 22-24
SPI slave, 22-27
and SPORT, 24-4
startup, 6-18
stop mode, 6-11, 6-69
stopping transfers, 6-29
support for peripherals, 1-6
switching peripherals from, 6-75
and synchronization with PPI, 15-13
synchronization, 6-51 to 6-62
synchronized transition, 6-28
termination without abort, 6-29
throughput, 6-43
traffic control, 6-49
traffic exceeding available bandwidth, 6-46
transfers, 1-11, 2-15
transfers, urgent, 6-46
transmit, 6-26
transmit operation, MAC, 21-23
transmit restart or finish, 6-35, 6-36
triggering transfers, 6-62
two descriptors in small list flow mode, example, 6-97
two-dimensional, 6-12
two-dimensional memory DMA setup example, 6-95
types supported, 1-11
and UART, 25-18, 25-28
using descriptor structures example, 6-99
variable descriptor size, 6-16
with PPI, 15-23
word size, changing, 6-28
work units, 6-14, 6-23, 6-25
DMA2D bit, 6-69, 6-72
DMA2D (DMA mode) bit, 8-6
DMA address high (DMA_ADDR_HIGH) bits, 26-146
DMA address low (DMA_ADDR_LOW) bits, 26-146
DMA_ADDR_HIGH (DMA address high) bits, 26-146
DMA_ADDR_LOW (DMA address low) bits, 26-146
DMA burst mode selection (BURST_MODE) bits, 26-144
DMA bus, *See* DAB
DMA bus error (BUSERROR) bit, 26-144
DMACFG field, 6-22, 6-63
DMA channel registers, 6-64
DMACODE, 17-121
DMA Code field DMACODE, 17-25
DMA complete (COMPLETE) bit, 8-29
DMA configuration (DMAx_CONFIG) registers, 6-68
DMA configuration (MDMA_yy_CONFIG) registers, 6-68
DMA controller, 6-2
DMA core bus, *See* DCB
DMA_COUNT_LOW (DMA count low) bits, 26-147, 26-148
DMA count low (DMA_COUNT_LOW) bits, 26-147, 26-148
DMA descriptor pairs, 21-24

DMA_DIR (DMA direction) bit, 8-29
DMA direction (DMA_DIR) bit, 8-29
DMA direction (WNR) bit, 6-69, 6-72, 8-6
DMA_DONE bit, 6-11, 6-74
DMA_DONE interrupt, 6-73
DMA enable (DMA_ENA) bit, 26-144
DMA_ENA (DMA enable) bit, 26-144
DMAEN bit, 6-19, 6-62, 6-69, 6-72
DMA_ERR bit, 6-11, 6-74
DMA_ERROR interrupt, 6-30
DMA error interrupts, 6-75
DMA external bus, *See* DEB
DMA mode 0/1 selection (MODE) bit, 26-144
DMA mode select (DMAREQMODE_R) bit, 26-121
DMA mode select (DMAREQMODE_RH) bit, 26-121
DMA mode select (DMAREQMODE_T) bit, 26-115
DMA performance optimization, 6-42
DMA queue completion interrupt, 6-61
DMA ready (READY) bit, 8-29
DMA registers, 6-64
DMAREQ_ENA_R (DMA request enable) bit, 26-121
DMAREQ_ENA_T (DMA request enable) bit, 26-115
DMAREQMODE_R (DMA mode select) bit, 26-121
DMAREQMODE_RH (DMA mode select) bit, 26-121
DMAREQMODE_T (DMA mode select) bit, 26-115
DMA request enable (DMAREQ_ENA_R) bit, 26-121
DMA request enable (DMAREQ_ENA_T) bit, 26-115
DMA_RUN bit, 6-22, 6-59, 6-63, 6-74
DMA_RUN bit), 6-11
DMARx pin, 6-38
DMA start address field, 6-76
DMA_TC_CNT (DMA traffic control counter) register, 6-92
DMA_TC_PER (DMA traffic control counter period) register, 6-48, 6-92
DMA traffic control registers, 6-90
DMA_TRAFFIC_PERIOD field, 6-92
DMA Tx or Rx selection (DIRECTION) bit, 26-144
DMAx_CONFIG (DMA configuration) registers, 6-8, 6-18, 6-25, 6-68
DMAx_CURR_ADDR (current address) registers, 6-76
DMAx_CURR_DESC_PTR (current descriptor pointer) registers, 6-83
DMAx_CURR_X_COUNT (current inner loop count) registers, 6-78, 6-79
DMAx_CURR_Y_COUNT (current outer loop count) registers, 6-81
DMAx_INT (USB DMA endpoint x interrupt) bits, 26-143
DMAx_IRQ_STATUS (interrupt status) registers, 6-73, 6-74
DMAx_NEXT_DESC_PTR (next descriptor pointer) registers, 6-18, 6-82
DMAx_PERIPHERAL_MAP (peripheral map) registers, 5-6, 6-68
DMAx_START_ADDR (start address) registers, 6-18, 6-76
DMAx_X_COUNT (inner loop count) registers, 6-77
DMAx_X_MODIFY (inner loop address increment) registers, 6-18, 6-79
DMAx_Y_COUNT (outer loop count) registers, 6-80
DMAx_Y_MODIFY (outer loop address increment) registers, 6-18, 6-81

DMC[1:0] field, 3-6
DMEM_CONTROL (data memory control) register, 3-4, 3-6
DNAK (data not acknowledged) bit, 23-35, 23-37
DOUBLE_FAULT, 17-100
DOUBLE_FAULT bit, 16-39
DOUBLE_RESET, 16-39
double word index[1:0] field, 3-7
DPMC, 18-2, 18-7
DR (data ready) bit, 25-12, 25-25
DR flag, 25-17
DRIVE_VBUS_OFF_ENA (drive VBUS off interrupt enable) bit, 26-136
drive VBUS off interrupt enable (DRIVE_VBUS_OFF_ENA) bit, 26-136
DRIVE_VBUS_ON_ENA (drive VBUS on interrupt enable) bit, 26-136
drive VBUS on interrupt enable (DRIVE_VBUS_ON_ENA) bit, 26-136
DRO (disable receive own frames) bit, 21-61, 21-62
DRQ[1:0] field, 6-47, 6-85, 6-86
DRTY (disable Tx retry on collision) bit, 21-61, 21-63
DRxPRI signal, 24-5
DRxPRI SPORT input, 24-6
DRxSEC signal, 24-5
DRxSEC SPORT input, 24-6
DSP libraries, 1-31
DTEST_COMMAND (data test command) register, 3-6
DTXCRC (disable automatic Tx CRC generation) bit, 21-61, 21-64
DTXPAD (disable automatic Tx padding) bit, 21-61, 21-64
DTxPRI signal, 24-5
DTxPRI SPORT output, 24-6
DTxSEC signal, 24-5
DTxSEC SPORT output, 24-6
dynamic power management, 1-26, 18-1
  controller, 18-2

E

EAB
  arbitration, 2-15
  clocking, 18-2
  and EBIU, 7-5
  frequency, 2-15
  performance, 2-15
early frame sync, 24-36
EAV signal, 15-5
EBC, 7-5
EBCAW[1:0] field, 7-62
EBE bit, 7-55, 7-61, 7-62
EBIU, 1-12, 7-1 to 7-80
  as slave, 7-5
  asynchronous interfaces supported, 7-1
  block diagram, 7-4
  bus errors, 7-7
  byte enables, 7-17
  clock, 7-1
  clocking, 18-2
  control registers, 7-6
  core transfers to SRAM, 7-23
  and DMA, 6-4
  error detection, 7-7
  external access extension, 7-15
  overview, 7-1
  programmable timing characteristics, 7-11
  read access period, 7-16
  request priority, 7-1
  shared pins, 7-6
  status register, 7-6
  system clock, 7-7
  wait states, 7-15

EBIU_AMBCTL0 (asynchronous memory bank control 0) register, 7-20, 7-21
EBIU_AMBCTL1 (asynchronous memory bank control 1) register, 7-20, 7-22
EBIU_AMGCTL (asynchronous memory global control) register, 7-20
EBIU_SDBCTL (SDRAM memory bank control) register, 7-61, 7-62
EBIU_SDGCTL (SDRAM memory bank control) register, 7-56
EBIU_SDGCTL (SDRAM memory global control) register, 7-65, 7-66
EBIU_SDRRC (SDRAM refresh rate control) register, 7-59
EBIU_SDSTAT (SDRAM control status) register, 7-74
EBSZ[2:0] field, 7-26, 7-61, 7-62
EBUFE bit, 7-58, 7-66, 7-72, 7-73
ECC0 (parity calculation result0) bits, 20-20
ECC1 (parity calculation result1) bits, 20-20
ECC2 (parity calculation result2) bits, 20-20
ECC3 (parity calculation result3) bits, 20-20
ECCCNT (transfer count) bits, 20-21
ECC_RST (ECC and NFC counters reset) bit, 20-21
ECINIT[15:0] field, 6-89
ECOUNT[15:0] field, 6-89
edge detection, GPIO, 9-17
EHR (enable host reads) bit, 8-26
EHW (enable host writes) bit, 8-26
elfloader.exe, 17-21
ELSI (enable Rx status interrupt) bit, 25-8, 25-28, 25-29
EMAC_ADDRHI (MAC address high) register, 21-52, 21-68
EMAC_ADDRLO (MAC address low) register, 21-52, 21-67
EMAC_FLC (MAC flow control) register, 21-52, 21-75
EMAC_HASHHI (EMAC multicast hash table high) register, 21-52, 21-69
EMAC_HASHLO (EMAC multicast hash table low) register, 21-52, 21-69
EMAC_MMC_CTL (MAC management counters control) register, 21-53, 21-121
EMAC_MMC_RIRQE (EMAC MMC RX interrupt enable) register, 21-53, 21-114
EMAC_MMC_RIRQS (EMAC MMC RX interrupt status) register, 21-53, 21-112
EMAC_MMC_TIRQE (EMAC MMC TX interrupt enable) register, 21-53, 21-118
EMAC_MMC_TIRQS (EMAC MMC TX interrupt status) register, 21-53, 21-116
EMAC multicast hash table high (EMAC_HASHHI) register, 21-69
EMAC multicast hash table low (EMAC_HASHLO) register, 21-69
EMAC_OPMODE (MAC operating mode) register, 21-51, 21-61
EMAC_RXC_ALIGN register, 21-54
EMAC_RXC_ALLFRM register, 21-56
EMAC_RXC_ALLOCT register, 21-56
EMAC_RXC_BROAD register, 21-55
EMAC_RXC_DMAOVF register, 21-54
EMAC_RXC_EQ64 register, 21-57
EMAC_RXC_FCS register, 21-54
EMAC_RXC_GE1024 register, 21-57
EMAC_RXC_LNERRI register, 21-55
EMAC_RXC_LNERRO register, 21-55
EMAC_RXC_LONG register, 21-55

EMAC_RXC_LT1024 register, 21-57
EMAC_RXC_LT128 register, 21-57
EMAC_RXC_LT256 register, 21-57
EMAC_RXC_LT512 register, 21-57
EMAC_RXC_MACCTL register, 21-56
EMAC_RXC_MULTI register, 21-55
EMAC_RXC_OCTET register, 21-54
EMAC_RXC_OK register, 21-54
EMAC_RXC_OPCODE register, 21-56
EMAC_RXC_PAUSE register, 21-56
EMAC_RXC_SHORT register, 21-57
EMAC_RXC_TYPED register, 21-56
EMAC_RXC_UNICST register, 21-54
EMAC_RX_IRQE (EMAC RX frame status interrupt enable) register, 21-53, 21-104
EMAC_RX_STAT (EMAC RX current frame status) register, 21-53, 21-94
EMAC_RX_STKY (EMAC RX sticky frame status) register, 21-53, 21-100
EMAC_STAADD (MAC station management address) register, 21-10, 21-52, 21-72
EMAC_STADAT (MAC station management data) register, 21-10, 21-52, 21-74
EMAC_SYSCTL (MAC system control) register, 21-10, 21-53, 21-90
EMAC_SYSTAT (MAC system status) register, 21-53, 21-91
EMAC_TXC_1COL register, 21-57
EMAC_TXC_ALLFRM register, 21-59
EMAC_TXC_ALLOCT register, 21-59
EMAC_TXC_BROAD register, 21-59
EMAC_TXC_CRSERR register, 21-58
EMAC_TXC_DEFER register, 21-58
EMAC_TXC_DMAUND register, 21-58
EMAC_TXC_EQ64 register, 21-59
EMAC_TXC_GT1COL register, 21-57
EMAC_TXC_LATECL register, 21-58
EMAC_TXC_LT128 register, 21-59
EMAC_TXC_MACCTL register, 21-59
EMAC_TXC_MULTI register, 21-59
EMAC_TXC_OCTET register, 21-58
EMAC_TXC_OK register, 21-57
EMAC_TXC_UNICST register, 21-58
EMAC_TXC_XS_COL register, 21-58
EMAC_TXC_XS_DFR register, 21-59
EMAC_TX_IRQE (EMAC TX frame status interrupt enable) register, 21-53, 21-112
EMAC_TX_STAT (EMAC TX current frame status) register, 21-53, 21-105
EMAC_TX_STKY (EMAC TX sticky frame status) register, 21-53, 21-109
EMAC_VLAN1 (MAC VLAN1 tag) register, 21-52, 21-77
EMAC_VLAN2 (MAC VLAN2 tag) register, 21-52, 21-77
EMAC_WKUP_CTL (MAC wakeup frame control and status) register, 21-52, 21-78
EMAC_WKUP_FFCMD (MAC wakeup frame filter commands) register, 21-52, 21-86
EMAC_WKUP_FFCRC0 (MAC wakeup frame filter CRC0/1) register, 21-52, 21-88
EMAC_WKUP_FFCRC1 (MAC wakeup frame filter CRC2/3) register, 21-52, 21-88
EMAC_WKUP_FFMSKx (MAC wakeup frameX byte mask) registers, 21-52, 21-81
EMAC_WKUP_FFOFF (EMAC wakeup frame filter offsets) register, 21-52, 21-88
EMISO (enable MISO) bit, 22-36, 22-37
EMREN bit, 7-48, 7-66, 7-73
EMRS command, 7-48

emulation, and timer counter, 10-42
EMU_RUN bit, 10-41, 10-42, 10-47
enable host reads (EHR) bit, 8-26
enable host writes (EHW) bit, 8-26
enable Pxn interrupt A bit, 9-36
enable Pxn interrupt B bit, 9-36
ENABLE_SUSPENDM (suspend mode output enable) bit, 26-98
enable wakeup filter 0 bit, 21-86, 21-87
enable wakeup filter 1 bit, 21-86, 21-87
enable wakeup filter 2 bit, 21-86, 21-87
enable wakeup filter 3 bit, 21-86
enabling
  interrupts, 5-5
ENDCPLB bit, 3-6
endpoint number (EPNUM) bits, 26-144
endpoint x Rx enable (EPx_RX_ENA) bits, 26-96
endpoint x Tx enable (EPx_TX_ENA) bits, 26-96
entire field mode, PPI, 15-9
EP0_NAK_LIMIT (EP0 NAK limit) bits, 26-128
EP0 NAK limit (EP0_NAK_LIMIT) bits, 26-128
EP0_RX_COUNT (received byte count in EP0 FIFO) bits, 26-126
EP halted after a NAK (NAK_TIMEOUT_H) bit, 26-111, 26-115
EPNUM (endpoint number) bits, 26-144
EPROM, 1-9
EPS (even parity select) bit, 25-22
EPx_RX_ENA (endpoint x Rx enable) bits, 26-96
EPx_RX_E (USB Rx endpoint x interrupt enable) bits, 26-106
EPx_RX (USB Rx endpoint x interrupt) bits, 26-104
EPx_TX_ENA (endpoint x Tx enable) bits, 26-96
EPx_TX_E (USB Tx endpoint x interrupt enable) bits, 26-105
EPx_TX (USB Tx endpoint x interrupt) bits, 26-103
ERBFI (enable receive buffer full interrupt) bit, 25-7, 25-12, 25-27, 25-28
ERMS command, 7-33
ERR_DET (error detected) bit, 15-31, 15-32
ERR_NCOR (error not corrected) bit, 15-32
ERROR_H (timeout error) bit, 26-111
ERROR_RH (timeout error indicator) bit, 26-121
errors
  DMA, 6-29
  not detected by DMA hardware, 6-31
  startup, and timers, 10-8
error signals, SPI, 22-40 to 22-42
ERROR_TH (timeout error indicator) bit, 26-115
ERR_TYP[1:0] field, 10-7, 10-40, 10-41, 10-47
ERR_TYP bits, 10-28
ETBEI (enable transmit buffer empty interrupt) bit, 25-6, 25-12, 25-18, 25-27, 25-28
Ethernet controller
  architecture, 21-3
Ethernet frame header, 21-50
Ethernet MAC, *See* MAC
event controller, 5-2
event handling, 5-2
events
  definition, 5-3
  types of, 5-2
event system, 5-3
event vector table (EVT), 5-2

EVT1 register, 17-8
EXT_CLK mode, 10-31 to 10-33, 10-43
  control bit and register usage, 10-46
  flow diagram, 10-32
extended mode register, initialization, 7-49
external
  emulator debugger, 10-42
  SDRAM memory, 7-26
external access bus, *See* EAB
external bank, SDRAM, 7-32
external bus interface unit, *See* EBIU
external crystal, 1-25
external memory, 1-9, 3-5
  design issues, 19-4
  interfaces, 7-6
external memory map, 3-1
  figure, 7-3
external PHY, 21-4
EXTEST instruction, B-6
EZ-KIT Lite card, 1-31

F

FAST (fast mode) bit, 23-32, 23-34
fast mode, TWI, 23-10
FBBRW (fast back-to-back read to write) bit, 7-66, 7-73
FDMODE (full duplex mode) bit, 21-61, 21-62
FE (framing error) bit, 25-25, 25-26
FFE (force framing error on transmit) bit, 25-32, 25-33
FIFO, 7-1
  asynchronous connection, 6-39
FIFOEMPTY bit, 8-29
FIFO flush (HOST_FLUSH) bit, 8-26
FIFOFULL bit, 8-29
FIFO_FULL_R (detected FIFO not empty) bit, 26-121
FIFO_NOT_EMPTY_T (data packet in FIFO indicator) bit, 26-115
filtering, MAC, 21-13
finish control command, DMA, 6-34, 6-35
flash
  interface, 19-5
  memory, 7-1
flash memory, 1-9
FLCBUSY (flow control busy status) bit, 21-75, 21-76
FLCE (flow control enable) bit, 21-16, 21-75, 21-76
FLCPAUSE[15:0] field, 21-75
FLD (field indicator) bit, 15-32, 15-33
FLD_SEL (active field select) bit, 15-4, 15-27, 15-30
flex descriptors, 6-3
FLGx (slave select value) bit, 22-38, 22-39
FLOW[2:0] field, 6-23, 6-24, 6-57, 6-69
flow charts
  DMA, 6-19, 6-20
  general-purpose timers interrupt structure, 10-6
  GPIO, 9-22
  GPIO interrupt generation, 9-19
  PPI, 15-25
  SPI core-driven, 22-30
  SPI DMA, 22-31
  timer EXT_CLK mode, 10-32
  timer PWM_OUT mode, 10-11
  timer WDTH_CAP mode, 10-24
  TWI master mode, 23-25
  TWI slave mode, 23-24
FLOW mode, DMA, 6-17
FLOW (next operation) bit, 6-15, 6-16, 8-6
FLOW value, DMA, 6-21
FLSx (slave select enable) bit, 22-8, 22-38
flush endpoint FIFO (FLUSHFIFO) bit, 26-111
flush endpoint FIFO (FLUSHFIFO_R) bit, 26-121

flush endpoint FIFO (FLUSHFIFO_T) bit, 26-115
FLUSHFIFO (flush endpoint FIFO) bit, 26-111
FLUSHFIFO_R (flush endpoint FIFO) bit, 26-121
FLUSHFIFO_T (flush endpoint FIFO) bit, 26-115
FORCE_DATATOGGLE_T (force endpoint data toggle) bit, 26-115
force endpoint data toggle (FORCE_DATATOGGLE_T) bit, 26-115
FORCE_MSEL (force PLL frequency multiplier) bits, 26-141
force PLL frequency multiplier (FORCE_MSEL) bits, 26-141
FPE bit, 25-32, 25-33
frame buffer, Ethernet, 21-124
FrameCheckSequenceErrors register, 21-54
framed serial transfers, characteristics, 24-32
framed/unframed data, 24-31
frame filter evaluation, MAC, 21-14
FRAME_NUMBER (USB frame number) bits, 26-109
FramesAbortedDueToXSColls register, 21-58
FramesLen1024_MaxReceived register, 21-57
FramesLen128_255Received register, 21-57
FramesLen256_511Received register, 21-57
FramesLen512_1023Received register, 21-57
FramesLen65_127Received register, 21-57
FramesLen65_127Transmitted register, 21-59
FramesLenEq64Received register, 21-57
FramesLenEq64Transmitted register, 21-59
FramesLenLt64Received register, 21-57
FramesLostDueToIntMAC RcvError register, 21-54
FramesLostDueToIntMACXmitError register, 21-58
FramesReceivedAll register, 21-56
FramesReceivedOK register, 21-54
frame start detect, PPI, 15-36
FramesTransmittedAll register, 21-59
FramesTransmittedOK register, 21-57
FramesWithDeferredXmissions register, 21-58
FramesWithExcessiveDeferral register, 21-59
frame sync
  active high/low, 24-34
  early, 24-36
  early/late, 24-36
  external/internal, 24-33
  internal, 24-26
  internally generated, 24-66
  late, 24-36
  multichannel mode, 24-19
  sampling edge, 24-34
  SPORT options, 24-31
frame sync divider[15:0] field, 24-66, 24-67
frame synchronization
  PPI in GP modes, 15-20
  and SPORT, 24-3
frame sync polarity, PPI and timer, 15-21
frame sync pulse
  use of, 24-52
  when issued, 24-52
frame sync signal, control of, 24-52, 24-57
FrameTooLongErrors register, 21-55
frame track error, 15-32, 15-36

frequencies, clock and frame sync, 24-26
frequency, DEB, 2-15
frequency, EAB, 2-15
FSDEV (full- or high-speed device indicator) bit, 26-132, 26-134
FSDR (frame sync to data relationship) bit, 24-22, 24-68
FS_EOF1 (full-speed EOF 1) bits, 26-138
F signal, 15-33
FT_ERR (frame track error) bit, 15-32, 15-36
full duplex, 24-4, 24-6
  SPI, 22-2
FULL_ON bit, 18-28
full-on mode, 1-26, 18-8
full- or high-speed device indicator (FSDEV) bit, 26-132, 26-134
full-speed EOF 1 (FS_EOF1) bits, 26-138
FUNCTION_ADDRESS (USB peripheral device address) bits, 26-100
function enable (PORTF_FER) register, 9-10
function enable (PORTG_FER) register, 9-10
function enable (PORTH_FER) register, 9-10
function enable (PORTx_FER) registers, 9-30

G

GAIN[1:0] field, 18-29, 18-30
GCALL (general call) bit, 23-30, 23-31
general call address, TWI, 23-10
general-purpose interrupts, 5-2, 5-3
general-purpose I/O
  overview, 1-13
general-purpose I/O, *See* GPIO
general-purpose ports, 1-13, 9-1 to 9-43
  assigning interrupt channels, 9-17
  interrupt channels, 9-17
  interrupt generation flow, 9-17
  latency, 9-12
  pin defaults, 9-3
  pins, interrupt, 9-16
  throughput, 9-11
general-purpose ports, See GPIO
general-purpose timers, 10-1 to 10-57
  aborting immediately, 10-22
  and startup errors, 10-8
  autobaud mode, 10-31
  block diagram, 10-58
  buffer registers, 10-44
  capture mode, 10-5
  clock source, 10-3
  code examples, 10-48
  control bit summary, 10-46
  counter, 10-4
  disable timing, 10-22
  enabling, 10-5, 10-33
  error detection, 10-7
  EXT_CLK mode, 10-43
  external interface, 10-3
  features, 10-2
  flow diagram for EXT_CLK mode, 10-32
  generating maximum frequency, 10-15
  illegal states, 10-7, 10-9
  internal interface, 10-4
  internal timer structure, 10-3
  interrupts, 10-4, 10-5, 10-14, 10-28
  interrupt setup, 10-50
  interrupt structure, 10-6
  measurement report, 10-25, 10-26, 10-27
  non-overlapping clock pulses, 10-54
  output pad disable, 10-12
  overflow, 10-4

general-purpose timers *(continued)*
periodic interrupt requests, 10-51
port setup, 10-48
and PPI, 10-59
preventing errors in PWM_OUT mode, 10-44
programming model, 10-33
PULSE_HI toggle mode, 10-15
PWM mode, 10-5
PWM_OUT mode, 10-10 to 10-23, 10-43
registers, 10-34
signal generation, 10-49
single pulse generation, 10-12
size of register accesses, 10-35
stopping in PWM_OUT mode, 10-21
three timers with same period, 10-17
two timers with non-overlapping clocks, 10-18
waveform generation, 10-13
WDTH_CAP mode, 10-23, 10-43
WDTH_CAP mode configuration, 10-56
WDTH_CAP mode flow diagram, 10-24
GEN (general call enable) bit, 23-28, 23-29
glitch filtering, UART, 25-10
GLOBAL_ENA (USB enable) bit, 26-96
glueless connection, 19-5
GM (get more data) bit, 22-21, 22-37
GPIO, 1-13, 9-1 to 9-43
assigned to same interrupt channel, 9-20
clearing interrupt conditions, 9-18
clear registers, 9-15
code examples, 9-42
configuration, 9-13
data registers, 9-13, 9-14, 9-15
direction registers, 9-13
edge detection, 9-17
edge-sensitive, 9-14
GPIO, 1-13, 9-1 to 9-43 *(continued)*
flow chart, 9-22
function enable registers, 9-12, 9-16
input buffers, 9-13
input driver, 9-14
input enable registers, 9-13, 9-16
interrupt channels, 9-21
interrupt generation flow chart, 9-19
interrupt request, 5-14
interrupts, 9-17
interrupt sensitivity registers, 9-17
mask data registers, 9-18
mask interrupt clear registers, 9-20
mask interrupt set registers, 9-18
mask interrupt toggle registers, 9-20
mask registers, 9-18
overview, 1-13
pins, 9-12, 9-13
polarity registers, 9-16
port F, 9-5
registers, 9-27
set registers, 9-15
toggle registers, 9-16
using as input, 9-13
write operations, 9-14
writes to registers, 9-15
GPIO clear (PORTxIO_CLEAR) registers, 9-33
GPIO data (PORTxIO) registers, 9-32
GPIO direction (PORTxIO_DIR) registers, 9-31
GPIO input enable (PORTxIO_INEN) registers, 9-32
GPIO mask interrupt A clear registers, 9-39
GPIO mask interrupt A (PORTxIO_MASKA_CLEAR) registers, 9-36
GPIO mask interrupt A set (PORTxIO_MASKA_SET) registers, 9-37

GPIO mask interrupt A toggle (PORTxIO_MASKA_TOGGLE) registers, 9-41
GPIO mask interrupt B clear (PORTxIO_MASKB_CLEAR) registers, 9-40
GPIO mask interrupt B (PORTxIO_MASKB) registers, 9-36
GPIO mask interrupt B set (PORTxIO_MASKB_SET) registers, 9-38
GPIO mask interrupt B toggle (PORTxIO_MASKB_TOGGLE) registers, 9-42
GPIO pins, 9-12
GPIO polarity (PORTxIO_POLAR) registers, 9-34
GPIO set on both edges (PORTxIO_BOTH) registers, 9-35
GPIO set (PORTxIO_SET) registers, 9-33
GPIO toggle (PORTxIO_TOGGLE) registers, 9-34
GP modes, PPI, 15-14
ground plane, 19-9
GUWKE (global unicast wakeup enable) bit, 21-79

H

H.100, 24-22
H.100 standard protocol, 24-25
handshake DMA, 1-12
handshake MDMA, 6-9
  interrupts, 6-41
handshake MDMA configuration (HMDMAx_BCINIT) registers, 6-38
handshake MDMA control (HMDMAx_CONTROL) registers, 6-84
handshake MDMA control registers, 6-86
handshake MDMA current block count (HMDMAx_BCOUNT) registers, 6-38, 6-87
handshake MDMA current block count registers (HMDMAx_BCOUNT), 6-88
handshake MDMA current edge count (HMDMAx_ECOUNT) registers, 6-38, 6-39, 6-88, 6-89
handshake MDMA edge count overflow interrupt (HMDMAx_ECOVERFLOW) registers, 6-90
handshake MDMA edge count urgent (HMDMAx_ECURGENT) registers, 6-89, 6-90
handshake MDMA initial block count (HMDMAx_BCINIT) registers, 6-87
handshake MDMA initial edge count (HMDMAx_ECINIT) registers, 6-39, 6-89
handshaking MDMA operation, 6-4
handshaking memory DMA (HMDMA), 6-2
hardware reset, 17-5, 17-6, 17-8
header checksum field
  HDRCHK, 17-28
header signature
  HDRSGN, 17-29
hibernate state, 1-27, 7-45, 18-11
high-frequency design considerations, 19-7
high- or full-speed device indicator (FSDEV) bit, 26-132, 26-134
high-speed EOF 1 (HS_EOF1) bits, 26-138
high speed mode enable (HS_ENABLE) bit, 26-98
high speed mode flag (HS_MODE) bit, 26-98
HIRQ (host interrupt request) bit, 8-29

HMDMA, 6-2, 6-9
HMDMAEN bit, 6-37, 6-39, 6-86
HMDMAx_BCINIT (handshake MDMA configuration) registers, 6-38, 6-87
HMDMAx_BCOUNT (handshake MDMA current block count) registers, 6-38, 6-87, 6-88
HMDMAx_CONTROL (handshake MDMA control) registers, 6-84, 6-86
HMDMAx_ECINIT (handshake MDMA initial edge count) registers, 6-39, 6-89
HMDMAx_ECOUNT (handshake MDMA current edge count) registers, 6-38, 6-39, 6-88, 6-89
HMDMAx_ECOVERFLOW (handshake MDMA edge count overflow interrupt) registers, 6-90
HMDMAx_ECURGENT (handshake MDMA edge count urgent) registers, 6-89, 6-90
HM (hash filter multicast addresses) bit, 21-61, 21-66
HMVIP, 24-25
hold, for EBIU asynchronous memory controller, 7-19
horizontal blanking, 15-6
horizontal tracking, PPI, 15-33
host acknowledge mode timeout (HOST_TIMEOUT) register, 8-31
HOST_CONFIG (host configuration) word, 8-6
HOST_CONFIG (host configuration word) register, 8-6
host configuration word (HOST_CONFIG) register, 8-6
HOST_CONTROL (host control) register, 8-26
host control (HOST_CONTROL) register, 8-26
host DMA port
  multiplexing concern, 8-3
  overview, 8-1
HOSTDP, 8-1
  block diagrams, 8-4
  configuration, 8-5
  functional blocks, 8-4
  interface pins, 8-3
  operation, 8-3
HOSTDP controller
  features, 8-2
host enable (HOST_EN) bit, 8-26
HOST_END (host endianess) bit, 8-26
host endianess (HOST_END) bit, 8-26
HOST_EN (host enable) bit, 8-26
HOST_FLUSH (FIFO flush) bit, 8-26
host handshake (HSHK) bit, 8-29
host interrupt request (HIRQ) bit, 8-29
host negotiation request (HOST_REQ) bit, 26-132, 26-134
host ready override (HRDY_OVR) bit, 8-26
HOST_REQ (host negotiation request) bit, 26-132, 26-134
host reset (HOST_RST) bit, 8-26
HOST_RST (host reset) bit, 8-26
HOST_STATUS (host status) register, 8-29
host status (HOST_STATUS) register, 8-29
host timeout count (COUNT_TIMEOUT) bits, 8-31
HOST_TIMEOUT (host acknowledge mode timeout) register, 8-31
host timeout (TIMEOUT) bit, 8-29
hours[3:0] field, 14-20, 14-22
hours[4] bit, 14-20, 14-22
hours event flag bit, 14-21
hours interrupt enable bit, 14-20

HRDY_OVR (host ready override) bit, 8-26
HS_ENABLE (high speed mode enable) bit, 26-98
HS_EOF1 (high-speed EOF 1) bits, 26-138
HSHK (host handshake) bit, 8-29
HS_MODE (high speed mode flag) bit, 26-98
HU (hash filter unicast addresses) bit, 21-61, 21-66

I

I2C, *See* TWI
I^2C bus standard, 1-15, 23-2
I^2S, 1-18
  format, 24-11
  serial devices, 24-3
ICIE (illegal gray/binary code interrupt enable) bit, 13-21
ICII (illegal gray/binary code interrupt identifier) bit, 13-22
idle state
  waking from, 5-6
IEEE 1149.1 standard, *See* JTAG standard
IEEE 802.3, 1-16, 21-4, 21-8, 21-43
IFE (inverse filtering) bit, 21-61, 21-66
IMASK (interrupt mask) register
  initialization, 5-8
INCOMPTX_RH (large packet split) bit, 26-121
INCOMPTX_R (large packet split) bit, 26-121
INCOMPTX_T (large packet split) bit, 26-115
increase PLL charge pump current (TM_SELC) bit, 26-141
INIT bit, 17-38
initcall address/symbol command, 17-39
initcode routines, 17-37
initialization
  IMASK register, 5-8
  interrupt, 5-8
  SDRAM, 7-70
initializing
  DMA, 6-18
init initcode.dxe command, 17-39
inner loop address increment registers
  (DMAx_X_MODIFY), 6-79
  (MDMA_yy_X_MODIFY), 6-79
inner loop count registers
  (DMAx_X_COUNT), 6-77
  (MDMA_yy_X_COUNT), 6-77
INPDIS (CUD and CDZ input disable) bit, 13-20
input buffers, GPIO, 9-13
input clock, *See* CLKIN
input delay bit, 18-26, 18-27
input driver, GPIO, 9-14
InRangeLengthErrors register, 21-55
instruction bit scan ordering, B-5
instruction register (IR), B-2, B-4
instructions, 1-28
  private, B-4
  public, B-4
  See also instructions by name
INT_ENA (interrupt enable) bit, 26-144
interfaces
  external memory, 7-6
  internal memory, 7-5
  on-chip, 2-2
  overview, 2-3
  RTC, 14-3
  system, 2-1
inter IC bus, 23-2
interlaced video, 15-6
interleaving
  of data in SPORT FIFO, 24-59
  SPORT data, 24-7

internal
address mapping (table), 7-63
bank, 7-32, G-12
SDRAM banks, 7-27
internal boot ROM, 17-1
internal clocks, 2-3
internal/external frame syncs, *See* frame sync
internal memory, 1-9
accesses, 3-1
interfaces, 7-5
internal TSR register, UART, 25-7
internal voltage regulator, 1-24
interrupt
for peripheral, 5-1
interrupt conditions, UART, 25-29
interrupt enable (INT_ENA) bit, 26-144
interrupt handler and DMA synchronization, 6-59
interrupt mask (CNT_IMASK) register, 13-19, 13-21
interrupt mode (INT_MODE) bit, 8-26
interrupt output, SPI, 22-17
interrupt request lines, peripheral, 5-15
interrupts, 5-1 to 5-15
assigning priority for UART, 25-13
channels, assigning, 9-17
channels, GPIO, 9-17
clearing requests, 5-13
configuring and servicing, 19-2
configuring for MAC, 21-47
control of system, 5-2
default mapping, 5-3
definition, 5-3
determining source, 5-5
DMA channels, 5-6
DMA_ERROR, 6-30
DMA error, 6-75
DMA overflow, 6-42
DMA queue completion, 6-61
nterrupts, 5-1 to 5-15 *(continued)*
enabling, 5-5
Ethernet event, 21-39
ethernet event, 21-39
evaluation of GPIO interrupts, 9-20
igeneral-purpose, 5-2, 5-3
general-purpose timers, 10-4, 10-5, 10-14, 10-28
generated by peripherals, 5-8
GPIO, 9-16, 9-17, 9-21
handshake MDMA, 6-41
initialization, 5-8
inputs and outputs, 5-4
mapping, 5-4
mask function, 5-7
multiple sources, 5-9
peripheral, 5-2, 5-3, 5-4 to 5-7
prioritization, 5-4
processing, 5-1, 5-8
programming examples, 5-13 to 5-15
remote wakeup, 21-35
reset, 17-10
routing overview, 5-16, 5-17
RTC, 14-3, 14-7, 14-14
shared, 5-4
software, 5-3
SPI, 22-17, 22-47
SPORT error, 24-39
SPORT RX, 24-39, 24-63
SPORT TX, 24-39, 24-60
system, 5-1
to wake core from idle, 5-6
UART, 25-11
use in managing a descriptor queue, 6-58
interrupt sensitivity (PORTxIO_EDGE) registers, 9-35
interrupt service routine, determining source of interrupt, 5-5

interrupt status registers
(DMAx_IRQ_STATUS), 6-73, 6-74
(MDMA_yy_IRQ_STATUS), 6-73, 6-74
INT_MODE (interrupt mode) bit, 8-26
I/O interface to peripheral serial device, 24-4
I/O memory space, 1-9
I/O pins, general-purpose, 9-13
IP checksum, MAC, 21-19
IRCLK (internal receive clock select) bit, 24-55, 24-57
IrDA, 25-32
receiver, 25-9
transmitter, 25-9
IrDA SIR, 25-5
IREN (enable IdDA mode) bit, 25-32
IRFS (internal receive frame sync select) bit, 24-33, 24-55, 24-57
IR instruction register, B-2, B-4
IRPOL bit, 25-11
IRQ bit, 10-48
IRQ_ENA bit, 10-41, 10-46, 10-48
isochronous transfer enable (ISO_R) bit, 26-121
isochronous transfer enable (ISO_T) bit, 26-115
isochronous update enable (ISO_UPDATE) bit, 26-98
ISO_R (isochronous transfer enable) bit, 26-121
ISO_T (isochronous transfer enable) bit, 26-115
ISO_UPDATE (isochronous update enable) bit, 26-98
ISR
supporting multiple interrupt sources, 5-7, 5-18
ISR and multiple interrupt sources, 5-9
ISR for the MMC interrupt, structure, 21-45
ITCLK (internal transmit clock select) bit, 24-49, 24-51
ITFS (internal transmit frame sync select) bit, 24-20, 24-33, 24-49, 24-52
ITHR[15:0] field, 6-90
ITU-R 601/656, 1-16
ITU-R 601 recommendation, 15-16
ITU-R 656 modes, 15-5, 15-9, 15-30, 15-32
active video only submode, 15-9, 15-10
and DLEN field, 15-26
entire field submode, 15-9
frame start detect, 15-36
frame synchronization, 15-11
output, 15-11
SAV codes, 15-33
supported, 1-17
vertical blanking interval only submode, 15-9, 15-10

J

JTAG, 1-31, B-1, B-2, B-4

L

L1
data cache, 3-4
data memory, 1-9
data memory subbanks, 3-4
data SRAM, 3-4
instruction memory, 1-9, 3-3
memory and core, 2-4
memory and DMA controller, 6-5
scratchpad RAM, 1-9
L1 instruction memory
address alignment, 3-3
subbanks, 3-3

LARFS (late receive frame sync) bit, 24-36, 24-55, 24-58
large descriptor mode, DMA, 6-15
large model mode, DMA, 6-71
large packet split (INCOMPTX_R) bit, 26-121
large packet split (INCOMPTX_RH) bit, 26-121
large packet split (INCOMPTX_T) bit, 26-115
LateCollisions register, 21-58
late frame sync, 24-18, 24-36
latency
  DAB, 2-14
  DMA, 6-25
  general-purpose ports, 9-12
  powerup, 7-55
  SDC, 7-40
  SDRAM, 7-70
LATFS (late transmit frame sync) bit, 24-36, 24-49, 24-53
LB (internal loopback enable) bit, 21-61, 21-62
LCRTE (enable Tx retry on late collision) bit, 21-28, 21-61, 21-63
level shifters, 19-11
lines per frame (PPI_FRAME) register, 15-36
lines per frame register, 15-35
line terminations, SPORT, 24-9
linked list, DMA, 21-124
little endian byte order, 23-45
loader file, 17-21
load error indicator (DATAERROR_R) bit, 26-121
loader utility, 17-21
LOCKCNT[15:0] field, 18-28
locked transfers, DMA, 2-14
logic voltage, controlling, 18-19
loopback feature, PPI, 15-10
loopback mode, UART, 25-24
LOOP (loopback mode enable) bit, 25-24
LOSTARB (lost arbitration) bit, 23-35, 23-38
Lower PBS02 Half Page (PBS02L, Bits 63–0), 17-114
low-speed device indicator (LSDEV) bit, 26-132, 26-134
low-speed EOF 1 (LS_EOF1) bits, 26-139
LRFS (low receive frame sync select) bit, 24-13, 24-32, 24-34, 24-55, 24-58
LSBF (LSB first) bit, 22-37
LSDEV (low-speed device indicator) bit, 26-132, 26-134
LS_EOF1 (low-speed EOF 1) bits, 26-139
LT_ERR_OVR flag, 15-33
LT_ERR_OVR (horizontal tracking overflow error) bit, 15-32, 15-33
LT_ERR_UNDR flag, 15-34
LT_ERR_UNDR (horizontal tracking underflow error) bit, 15-32, 15-33
LTFS (low transmit frame sync select) bit, 24-20, 24-32, 24-34, 24-49, 24-53

M

MAC, 1-16, 21-1 to 21-128
  aborted frames, 21-16
  address filter evaluation, 21-13
  address setup, 21-126
  alternative descriptor structure, 21-26
  block diagram, 21-3
  clocking, 21-4
  code examples, 21-122
  configuration, 21-46
  configuring address registers, 21-48
  configuring interrupts, 21-47
  configuring PHY, 21-49
  continuous polling, 21-44
  control frames, 21-16

MAC, 1-16, 21-1 to 21-128 *(continued)*
CRC-16 hash value calculation, 21-38
CRC-32 address calculation, 21-70
CRC state, 21-36
CSMA/CD protocol, 21-3
descriptor structure, alternative, 21-26
discarded frames, 21-15
DMA configuration, 21-123
DMA data transfer, 21-50
DMA descriptor pairs, 21-12, 21-24
DMA descriptors, 21-124
Ethernet event interrupts, 21-39
ethernet event interrupts, 21-39
Ethernet frame buffer, 21-124
Ethernet frame header, 21-50
features, 21-2
filters, 21-13
flexible descriptor structure, 21-26
frame filter evaluation, 21-14
frame reception and filtering, 21-13
internal interface, 21-7
IP checksum, 21-19
late collisions, 21-28
linked list of DMAs, 21-124
Magic Packet detection, 21-34
management counters, 21-43
MII management interface, 21-8
MMC interrupt, 21-40, 21-45
multiplexed pins, 21-46
multiplexing, 21-5
operation in sleep state, 21-32
overview, 1-16
peripheral, 21-10
PHY control routines, 21-126
PHYINT interrupt, 21-40
pins, 21-6
power management states, 21-7
protocol compliance, 21-8
read access to the PHY, 21-128
receive DMA operation, 21-11
receiving data, 21-50
registers, table, 21-51
remote wakeup frame filters, 21-34
RX/TX frame status interrupt operation, 21-42
RX automatic pad stripping, 21-17
RX DMA buffer structure, 21-18
RX DMA direction error detected, 21-41
RX DMA direction errors, 21-22
RX frame status buffer, 21-19
RX frame status classification, 21-19
RX frame status interrupt, 21-40
RX frame status register operation, 21-42
RX IP frame checksum calculation, 21-21
RX receive status priority, 21-20
speculative read, 21-43
station management read, 21-9
station management read transfer, 21-49
station management transfer done, 21-41
station management write, 21-9
station management write transfer, 21-48
transfer frame protocol, 21-9
transmit DMA operation, 21-23
transmitting data, 21-51
TX DMA data alignment, 21-27
TX DMA direction error detected, 21-41
TX DMA direction errors, 21-29
TX frame status classification, 21-28
TX frame status interrupt, 21-40
TX frame status register operation, 21-43
TX transmit status priority, 21-29
type definitions, 21-123
wake from hibernate, 21-30
wake from sleep, 21-31
wakeup frame detected, 21-41
MAC address high[15:0] field, 21-68
MAC address high (EMAC_ADDRHI) register, 21-68

MAC address low[15:0] field, 21-68
MAC address low[31:16] field, 21-68
MAC address low (EMAC_ADDRLO) register, 21-67
MACControlFramesReceived register, 21-56
MACControlFramesTransmitted register, 21-59
MAC flow control (EMAC_FLC) register, 21-75
MAC management counters control (EMAC_MMC_CTL) register, 21-121
MAC operating mode (EMAC_OPMODE) register, 21-61
MAC station management address (EMAC_STAADD) register, 21-72
MAC station management data (EMAC_STADAT) register, 21-74
MAC system control (EMAC_SYSCTL) register, 21-90
MAC system status (EMAC_SYSTAT) register, 21-91
MAC VLAN1 tag (EMAC_VLAN1) register, 21-77
MAC VLAN2 tag (EMAC_VLAN2) register, 21-77
MADDR[6:0] field, 23-35
magic packet, 21-34
MASK_BUSYIRQ (mask not busy IRQ) bit, 20-19
mask not busy IRQ (MASK_BUSYIRQ) bit, 20-19
MASK_RDRDY (mask read data ready) bit, 20-19
mask read data ready (MASK_RDRDY) bit, 20-19
MASK_WBEDGE (mask write buffer edge detect) bit, 20-19
MASK_WBOVF (mask write buffer overflow) bit, 20-19
MASK_WRDONE (mask write done) bit, 20-19
mask write buffer edge detect (MASK_WBEDGE) bit, 20-19
mask write buffer overflow (MASK_WBOVF) bit, 20-19
mask write done (MASK_WRDONE) bit, 20-19
masters
  DAB, 2-13
  PAB, 2-6
MAXCIE (max count interrupt enable) bit, 13-21
MAXCII (max count interrupt identifier) bit, 13-22
max count interrupt identifier (MAXCII) bit, 13-22
maximal count (CNT_MAX) register, 13-19, 13-26
maximum count interrupt enable (MAXCIE) bit, 13-21
maximum individual packet size (MaxPktSize), 26-32, 26-33, 26-34, 26-35, 26-36, 26-37
MAX_PACKET_SIZE_R (USB max Rx data in frame) bits, 26-120
MAX_PACKET_SIZE_T (USB max Tx data in frame) bits, 26-110
MaxPktSize (maximum individual packet size), 26-32, 26-33, 26-34, 26-35, 26-36, 26-37
MBDI bit, 6-41, 6-86
MCCRM[1:0] field, 24-68
MCDRXPE (multichannel DMA receive packing) bit, 24-68
MCDTXPE (multichannel DMA transmit packing) bit, 24-68

MCMEN (multichannel frame mode enable) bit, 24-18, 24-68
MCOMP (master transfer complete) bit, 23-42, 23-43
MCOMPM (master transfer complete interrupt mask) bit, 23-42
MDCDIV[5:0] field, 21-90
MDIO station management interface, 21-8
MDIR (master transfer direction) bit, 23-32, 23-34
MDMA channels, 6-7
MDMA controllers, 6-7
MDMA_ROUND_ROBIN_COUNT[4:0] field, 6-49, 6-92
MDMA_ROUND_ROBIN_PERIOD field, 6-48, 6-49, 6-92
MDMA_yy_CONFIG (DMA configuration) registers, 6-68
MDMA_yy_CURR_ADDR (current address) registers, 6-76
MDMA_yy_CURR_DESC_PTR (current descriptor pointer) registers, 6-83
MDMA_yy_CURR_X_COUNT (current inner loop count) registers, 6-78, 6-79
MDMA_yy_CURR_Y_COUNT (current outer loop count) registers, 6-81
MDMA_yy_IRQ_STATUS (interrupt status) registers, 6-73, 6-74
MDMA_yy_NEXT_DESC_PTR (next descriptor pointer) registers, 6-82
MDMA_yy_PERIPHERAL_MAP (peripheral map) registers, 6-68
MDMA_yy_START_ADDR (start address) registers, 6-76
MDMA_yy_X_COUNT (inner loop count) registers, 6-77
MDMA_yy_X_MODIFY (inner loop address increment) registers, 6-79
MDMA_yy_Y_COUNT (outer loop count) registers, 6-80
MDMA_yy_Y_MODIFY (outer loop address increment) registers, 6-81
measurement report, general-purpose timers, 10-25, 10-26, 10-27
memory, 3-1 to 3-7
  accesses to internal, 3-1
  architecture, 1-7, 3-1
  asynchronous, 3-1
  asynchronous interface, 19-5
  asynchronous region, 7-2
  boot ROM, 3-5
  configurations, 1-8
  external, 1-9, 3-5
  external memory, 7-6
  external memory map, 3-1
  external SDRAM, 7-26
  flash, 1-9
  internal, 1-9
  internal bank, 7-32, G-12
  I/O space, 1-9
  L1, 2-4
  L1 data, 1-9, 3-4
  L1 data cache, 3-4
  L1 instruction, 1-9, 3-3
  L1 scratchpad RAM, 1-9
  moving data between SPORT and, 24-39
  off-chip, 1-8, 1-9
  on-chip, 1-7, 1-9
  OTP, 1-10
  SDRAM, 3-1
  start locations of L1 instruction memory subbanks, 3-3
  structure, 1-7
  unpopulated, 7-9
memory conflict, DMA, 6-50

memory DMA, 1-12, 6-7
bandwidth, 6-45
buffers, 6-8
channels, 6-7
descriptor structures, 6-63
handshake operation, 6-9
priority, 6-48
scheduling, 6-48
timing, 6-45
transfer operation, starting, 6-8
transfer performance, 2-15, 2-16
transfers, 6-2, 6-5
word size, 6-8
memory map
ADSP-BF522, 3-1
ADSP-BF525, 3-1
ADSP-BF527, 3-1
memory map, external (figure), 7-3
memory-mapped registers, *See* MMRs
memory-to-memory transfers, 6-7
MEN (master mode enable) bit, 23-32, 23-34
MERR (master transfer error) bit, 23-42, 23-43
MERRM (master transfer error interrupt mask) bit, 23-42
MFD[3:0] field, 24-21, 24-68
MII
communications protocol, setting up, 21-48
management interface, 21-8
multiplexing, 21-5
pins, 21-6
MINCIE (min count interrupt enable) bit, 13-21
MINCII (min count interrupt identifier) bit, 13-22
min count interrupt identifier (MINCII) bit, 13-22
minimal count (CNT_MIN) register, 13-19, 13-26
minimum count interrupt enable (MINCIE) bit, 13-21
minutes[5:0] field, 14-20, 14-22
minutes event flag bit, 14-21
minutes interrupt enable bit, 14-20
MISO pin, 22-5, 22-12, 22-15, 22-16, 22-21
MMC, continuous polling, 21-44
MMCE (MMC counter enable) bit, 21-43, 21-121
MMC interrupt, 21-40, 21-45
MMCINT (MMC counter interrupt status) bit, 21-92, 21-93
MMRs, 1-9
address range, A-3
for PPI, 15-26
memory-related, 3-5
width, A-3
MODE (DMA mode 0/1 selection) bit, 26-144
mode fault error, 22-17, 22-41
modes
broadcast, 22-9, 22-15, 22-16
multichannel, 24-15
serial port, 24-11
SPI master, 22-15, 22-18
SPI slave, 22-16, 22-20
UART DMA, 25-18
UART non-DMA, 25-16
MODF (mode fault error) bit, 22-40, 22-41
MOSI pin, 22-5, 22-12, 22-15, 22-16, 22-21
moving data, serial port, 24-39
MPKE (magic packet wakeup enable) bit, 21-34, 21-79, 21-80
MPKS (magic packet received status) bit, 21-34, 21-79

M (PLL multiplier select) bits, 26-141
MPROG (master transfer in progress) bit, 23-35, 23-38
MRS command, 7-33, 7-47
MSEL[5:0] field, 18-3, 18-4, 18-26, 18-27
MSTR (master) bit, 22-36, 22-37
multibank operation, 7-42
MulticastFramesReceivedOK register, 21-55
MulticastFramesXmittedOK register, 21-59
multichannel frame, 24-20
multichannel frame delay field, 24-21
multichannel mode, 24-15
  enable/disable, 24-18
  frame syncs, 24-19
  SPORT, 24-19
multichannel operation, SPORT, 24-15 to 24-25
MultipleCollisionFrames register, 21-57
multiple interrupt sources, 5-9
multiple slave SPI systems, 22-8
multiplexed SDRAM addressing scheme figure, 7-27
multiplexing, 9-1
  MII and RMII, 21-5
MVIP-90, 24-25

N

NAK (not acknowledge) bit, 23-28, 23-29
NAK_TIMEOUT_H (EP halted after a NAK) bit, 26-111, 26-115
NAND address (NFC_ADDR) register, 20-15, 20-23
NAND command (NFC_CMD) register, 20-15, 20-24
NAND control (NFC_CTL) register, 20-15, 20-16
NAND data read (NFC_DATA_RD) register, 20-15, 20-25
NAND data width (NWIDTH) bit, 20-16
NAND data write (NFC_DATA_WR) register, 20-15, 20-24
NAND ECC count (NFC_COUNT) register, 20-15, 20-21
NAND ECC reset (NFC_RST) register, 20-15, 20-21
NAND interrupt mask (NFC_IRQMASK) register, 20-15, 20-19
NAND interrupt status (NFC_IRQSTAT) register, 20-15, 20-18
NAND page control (NFC_PGCTL) register, 20-15, 20-22
NAND read data (NFC_READ) register, 20-15, 20-23
NAND status (NFC_STAT) register, 20-15, 20-17
NBUSYIRQ (not busy IRQ) bit, 20-18
NBUSY (not busy) bit, 20-17
NDPH bit, 6-21
NDPL bit, 6-21
NDSIZE[3:0] field, 6-16, 6-69, 6-71
  legal values, 6-32
next descriptor pointer registers
  (DMAx_NEXT_DESC_PTR), 6-82
  (MDMA_yy_NEXT_DESC_PTR), 6-82
NFC_ADDR (NAND address) register, 20-15, 20-23
NFC_CMD (NAND command) register, 20-15, 20-24
NFC_COUNT (NAND ECC count) register, 20-15, 20-21
NFC_CTL (NAND control) register, 20-15, 20-16
NFC_DATA_RD (NAND data read) register, 20-15, 20-25
NFC_DATA_WR (NAND data write) register, 20-15, 20-24

NFC_ECC0 (NAND ECC 0) register, 20-15, 20-20
NFC_ECC1 (NAND ECC 1) register, 20-15, 20-20
NFC_ECC1 (NAND ECC 2) register, 20-20
NFC_ECC2 (NAND ECC 2) register, 20-15
NFC_ECC3 (NAND ECC 3) register, 20-15, 20-20
NFC_ECCx (NAND ECC) registers, 20-15, 20-20
NFC_IRQMASK (NAND interrupt mask) register, 20-15, 20-19
NFC_IRQSTAT (NAND interrupt status) register, 20-15, 20-18
NFC_PGCTL (NAND page control) register, 20-15, 20-22
NFC_READ (NAND read data) register, 20-15, 20-23
NFC_RST (NAND ECC reset) register, 20-15, 20-21
NFC_STAT (NAND status) register, 20-15, 20-17
NINT (pending interrupt) bit, 25-29, 25-30
NOP (no operation) command, 7-53
NOPREBOOT, 16-47, 16-48
normal frame sync mode, 24-36
normal timing, serial port, 24-36
not busy IRQ (NBUSYIRQ) bit, 20-18
no TxPktRdy for IN token (OVERRUN_R) bit, 26-121
no TxPktRdy for IN token (UNDERRUN_T) bit, 26-115
NTSC systems, 15-6
NWIDTH (NAND data width) bit, 20-16

O

OctetsReceivedAll register, 21-56
OctetsReceivedOK register, 21-54
OctetsTransmittedAll register, 21-59
OctetsTransmittedOK register, 21-58
OE (overrun error) bit, 25-25
off-chip
  bus connections, 2-8
  infrared driver, 25-9
  line drivers, 25-7
  memory, 1-8
  peripherals, 6-2
  signals, 9-16
off-chip memory, 1-9
  external access bus (EAB), 2-14
off-core
  accesses, 2-5
offsets, DMA descriptor elements, 6-16
OI bit, 6-86
OIE bit, 6-86
on-chip
  buses, 2-8
  internal voltage regulator, 1-24
  I/O devices, 1-9
  memory, 1-7, 1-9
  peripherals, 1-10, 6-2
  PLL, 1-25
  SDRAM interface controller, 7-46
on-chip voltage regulator, 1-27
one-time-programmable (OTP) memory, 1-10
open drain drivers, 22-2
open drain outputs, 22-15
open page, G-1
  in memory, 7-34

operating modes, 18-8
active, 1-26, 18-9
deep sleep, 1-27, 18-10
full-on, 1-26, 18-8
hibernate state, 1-27, 18-11
PPI, 15-4
sleep, 1-26, 18-9
transition, 18-11, 18-12
optimization, of DMA performance, 6-42
oscilloscope probes, 19-11
OTP_EBIU_AMBCTL, 17-111
OTP_EBIU_AMG, 17-110
OTP_EBIU_FCTL, 17-111
OTP_EBIU_POWERON_DUMMY_WRITE, 17-114
OTP_EBIU_SDRRC, 17-114
OTP_ENA_CLKOUT, 17-107
OTP_INVALID, 17-107
OTP_LOAD_PAGE17H, 17-107
OTP_LOAD_PAGE17L, 17-107
OTP memory
bfrom_OtpCommand (), 4-10
BFROM_OTP_READ, 4-20
bfrom_OtpRead(), 4-10
BFROM_OTP_WRITE, 4-21
bfrom_OtpWrite(), 4-10
BFROM_TOP_COMMAND, 4-17
error correction, 4-7
map, 4-3
overview, 4-1
OTP_NFC_CTL, 17-112
OTP_PLL_CTL, 17-109
OTP_PLL_DIV, 17-108
OTP_SET_BMODES, 17-107
OTP_SET_CALIB, 17-107
OTP_SET_EBIU_ASYNC, 17-107
OTP_SET_EBIU_SYNC, 17-107
OTP_SET_FCTL, 17-110
OTP_SET_MODE, 17-110
OTP_SET_PLL, 17-107
OTP_SET_VR, 17-107
OTP_SPI_BAUD, 17-108
OTP_SPI_FASTREAD, 17-108
OTP_START_PAGE, 17-112
OTP_TWI_CLKDIV, 17-108
OTP_TWI_PRESCALE, 17-108
OTP_TWI_TYPE, 17-107
OTP_VR_CTL, 17-109
OUT_DIS bit, 10-40, 10-41, 10-46, 10-59
outer loop address increment registers (DMAx_Y_MODIFY), 6-81
(MDMA_yy_Y_MODIFY), 6-81
outer loop count registers (DMAx_Y_COUNT), 6-80
(MDMA_yy_Y_COUNT), 6-80
OutOfRangeLengthField register, 21-55
output delay bit, 18-26, 18-27
output pad disable, timer, 10-12
overflow interrupt, DMA, 6-42
OVERRUN_R (no TxPktRdy for IN token) bit, 26-121

P

PAB, 2-6
arbitration, 2-6
bus agents (masters, slaves), 2-6
clocking, 18-1
and EBIU, 7-5
performance, 2-7
PACK_EN (packing mode enable) bit, 15-27, 15-28
packet transaction status (STATUSPKT_H) bit, 26-111
packing, serial port, 24-24
page 0x14, 17-110
page hit, and SDC, 7-40
page miss, 7-51
page miss, and SDC, 7-40
page read pending (PG_RD_STAT) bit, 20-17

page read start (PG_RD_START) bit, 20-22
page size, 7-63
page write done (WR_DONE) bit, 20-18
page write pending (PG_WR_STAT) bit, 20-17
page write start (PG_WR_START) bit, 20-22
PAL systems, 15-6
PAM (pass all multicast mode) bit, 21-61, 21-66
parallel peripheral interface, *See* PPI
parity calculation result 0 (ECC0) bits, 20-20
parity calculation result 1 (ECC1) bits, 20-20
parity calculation result 2 (ECC2) bits, 20-20
parity calculation result 3 (ECC3) bits, 20-20
PASR[1:0] field, 7-66, 7-67
PASR feature, 7-27
PAUSEMACCtrlFramesReceived register, 21-56
PBF (pass bad frames) bit, 21-61, 21-65
PBS01H, Bits 15–0(Upper PBS01 Half Page), 17-113
PBS01H, Bits 63–16(Upper PBS01 Half Page), 17-112
PBS02L, Bits 63–0(Lower PBS02 Half Page), 17-114
PC133 SDRAM controller, 1-12
PCF (pass control frames) bit, 21-15, 21-75, 21-76
PDWN bit, 18-26, 18-27
PEN (parity enable) bit, 25-22
PE (parity error) bit, 25-25
performance
  DAB, 2-14
  DCB, 2-14
  DEB, 2-14, 2-15
  DMA, 6-44
  EAB, 2-15
  general-purpose ports, 9-11
  memory DMA, 6-45
  memory DMA transfers, 2-15, 2-16
  optimization, DMA, 6-42
  PAB, 2-7
  SDRAM, 7-30
PERIOD_CNT bit, 10-12, 10-20, 10-25, 10-41, 10-46
period value[15:0] field, 11-6
period value[31:16] field, 11-6
peripheral
  DMA, 6-6, 6-106
  DMA channels, 6-43
  DMA transfers, 6-2
  error interrupts, 6-75
  interrupt request lines, 5-15
  supporting interrupts, 5-1
peripheral access bus, *See* PAB
Peripheral bus
  errors generated by SPORT, 24-40
peripheral DMA start address registers, 6-76
peripheral interrupts, 5-2, 5-3, 5-4 to 5-7
peripheral map registers
  (DMAx_PERIPHERAL_MAP), 6-68
  (MDMA_yy_PERIPHERAL_MAP), 6-68
peripheral pins, default configuration, 9-12

peripherals, 1-5
and buses, 1-6
compatible with SPI, 22-3
and DMA controller, 6-32
DMA support, 1-6
enabling, 9-4
interrupt generated by, 5-8
interrupts, clearing, 5-13
level-sensitivity of interrupts, 5-15
list of, 1-5
mapping to DMA, 6-106
multiplexing, 9-1
remapping DMA assignment, 6-6
switching from DMA to non-DMA, 6-75
timing, 2-4
used to wake from idle, 5-6
PF0 pin, 9-15
PFx pin, 22-7
PG_RD_START (page read start) bit, 20-22
PG_RD_STAT (page read pending) bit, 20-17
PG_SIZE (page size) bit, 20-16
PG_WR_START (page write start) bit, 20-22
PG_WR_STAT (page write pending) bit, 20-17
phase locked loop, See PLL
PHY, 21-4
configuring, 21-49
control routines, 21-126
initialization, minimum requirements, 21-128
read access, 21-128
PHYAD[4:0] field, 21-73
PHYCLKOE bit, 21-47
PHYIE (PHYINT interrupt enable) bit, 21-90, 21-91
PHYINT interrupt, 21-40
PHYINT (PHYINT interrupt status) bit, 21-92, 21-93
pin, SDRAM, 7-54
pin information, 19-1
pins, 19-1
GPIO, 9-12
MAC, 21-6
multiplexing, 9-1
unused, 19-13
pin state during SDC commands (table), 7-46
pin terminations, SPORT, 24-9
pipeline, lengths of, 6-53
pipelining
DMA requests, 6-38
SDC supported, 7-73
PJSE bit, 9-28, 9-29
PLL, 18-1
active (enabled but bypassed) mode, 18-9
active mode, 18-9
applying power to the PLL, 18-13
block diagram, 18-3
BYPASS bit, 18-9
CCLK derivation, 18-3
changing clock ratio, 18-6
clock control, 18-1
clock dividers, 18-3
clocking to SDRAM, 18-10
clock multiplier ratios, 18-3
configuration, 18-3
control bits, 18-11
deep sleep mode, 18-10
design overview, 18-2
disabled, 18-13
divide frequency, 18-3
DMA access, 18-9
dynamic power management controller (DPMC), 18-7
enabled, 18-13

PLL, 18-1 *(continued)*
hibernate state, 18-11
interacting with DPMC, 18-2
and internal clocks, 2-4
maximum performance mode, 18-8
modification in active mode, 18-11
multiplier select (MSEL) field, 18-3
operating modes, operational characteristics, 18-8
operating mode transitions, 18-11, 18-14
PDWN bit, 18-11
PLL_OFF bit, 18-13
PLL status (table), 18-8
power domains, 18-16
power savings by operating mode (table), 18-8
registers, table, 18-25
removing power to the PLL, 18-13
RTC interrupt, 18-10
SCLK derivation, 18-1, 18-3
sleep mode, 18-9
STOPCK bit, 18-11
voltage control, 18-7, 18-19
PLLCLKOE (PLL clock output enable) bit, 26-141
PLL clock output enable (PLLCLKOE) bit, 26-141
PLL control (PLL_CTL) register, 18-3, 18-4, 18-25, 18-26, 18-27
PLL_CTL (PLL control) register, 18-3, 18-4, 18-25, 18-26, 18-27
PLL divide register, 2-4
PLL_DIV (PLL divide) register, 18-5, 18-25, 18-26
PLL_LOCKCNT (PLL lock count) register, 18-25, 18-28
PLL_LOCKED bit, 18-28
PLL multiplier select (M) bits, 26-141
PLL_OFF bit, 18-26, 18-27
PLL stable indicator (PLL_STABLE) bit, 26-141
PLL_STABLE (PLL stable indicator) bit, 26-141
PLL_STAT (PLL status) register, 18-25, 18-28
PLL VCO frequency, changing, 7-44
PMAP[3:0] field, 6-6, 6-46, 6-68, 6-106
polarity, GPIO, 9-17
POLC (polarity change) bit, 15-4, 15-26, 15-27
polling DMA registers, 6-52
POLS bit, 15-4, 15-26, 15-27
PORT_CFG[1:0] field, 15-4, 15-27, 15-30
port connection, SPORT, 24-7
PORT_DIR bit, 13-26
PORT_DIR (direction) bit, 15-4, 15-27, 15-30
PORT_EN (enable) bit, 15-27, 15-31
port F
GPIO, 9-5, 9-13
interrupt A channel, 9-20
peripherals, 9-2
structure, 9-4
PORTF_FER (function enable) register, 9-10
port G
GPIO, 9-13
interrupt A channel, 9-20
peripherals, 9-2, 9-5
structure, 9-5
PORTG_FER (function enable) register, 9-10
port H
GPIO, 9-13
and MAC, 21-46
peripherals, 9-3
structure, 9-7

PORTH_FER (function enable) register, 9-10
port J
  peripherals, 9-3
  structure, 9-9
port pins, 9-3, 22-39
port pins, test access, B-2
PORT_PREF0 bit, 3-6
PORT_PREF1 bit, 3-6
ports, 1-13
  See also ports by name
  overview, 1-13
port width, PPI, 15-28
PORTx_FER (function enable) registers, 9-4, 9-12, 9-16, 9-30
PORTxIO_BOTH (GPIO set on both edges) registers, 9-35
PORTxIO_CLEAR (GPIO clear) registers, 9-33
PORTxIO_DIR (GPIO direction) registers, 9-31
PORTxIO_EDGE (interrupt sensitivity) registers, 9-35
PORTxIO (GPIO data) registers, 9-32
PORTxIO_INEN (GPIO input enable) registers, 9-16, 9-32
PORTxIO_MASKA_CLEAR (GPIO mask interrupt A clear) registers, 9-20, 9-39
PORTxIO_MASKA (GPIO mask interrupt A) registers, 9-36
PORTxIO_MASKA_SET (GPIO mask interrupt A set) registers, 9-37
PORTxIO_MASKA_TOGGLE (GPIO mask interrupt A toggle) registers, 9-41
PORTxIO_MASKB_CLEAR (GPIO mask interrupt B clear) registers, 9-20, 9-40
PORTxIO_MASKB (GPIO mask interrupt B) registers, 9-36
PORTxIO_MASKB_SET (GPIO mask interrupt B set) registers, 9-38
PORTxIO_MASKB_TOGGLE (GPIO mask interrupt B toggle) registers, 9-42
PORTxIO_POLAR (GPIO polarity) registers, 9-34
PORTxIO_SET (GPIO set) registers, 9-33
PORTxIO_TOGGLE (GPIO toggle) registers, 9-34
PORTx_MUX (port multiplexer control) register, 9-4, 9-28, 9-29
PORTx_MUX (port multiplexer control) registers, 9-4, 9-10
power
  dissipation, 18-16
  domains, 18-16
  plane, 19-9
power management, 1-26, 18-1 to 18-36
power-up
  SDRAM, 7-47, 7-70
  sequence mode, 7-70
  start delay, 7-69
  start enable, 7-70
power-up latency, SDC, 7-55
PPI, 15-2 to 15-39
  active video only mode, 15-10
  block diagram, 15-3
  clearing DMA completion interrupt, 15-39
  clock input, 15-3
  configure DMA registers, 15-37
  configuring registers, 15-38
  control byte sequences, 15-8
  control signal polarities, 15-26
  data input modes, 15-14 to 15-17
  data movement, 15-9
  data output modes, 15-17 to 15-19

PPI, 15-2 to 15-39 *(continued)*
data width, 15-26
delay before starting, 15-34
DMA operation, 15-23
edge-sensitive inputs, 15-21
enabling, 15-31, 15-38
enabling DMA, 15-38
entire field mode, 15-9
external frame sync modes, 15-16
external frame syncs, 15-16, 15-18
features, 15-2
FIFO, 15-33
flow diagram, 15-25
frame start detect, 15-36
frame synchronization with ITU-R 656, 15-11
frame sync polarity with timer peripherals, 15-21
frame track error, 15-32, 15-36
general flow for GP modes, 15-14
general-purpose modes, 15-12
GP modes, 15-14
GP modes with frame synchronization, 15-20
GP output, 13-3, 13-5, 13-8, 15-20
hardware signalling, 15-16
horizontal tracking, 15-33
interlaced video, 15-6
internal frame sync modes, 15-16
internal frame syncs, 15-17
internal frame syncs modes, 15-19
ITU-R 601 recommendation, 15-16
ITU-R 656 modes, 15-5
ITU-R 656 output mode, 15-11
loopback feature, 15-10
memory-mapped registers, 15-26
multiplexed with general-purpose timers, 10-59
no frame syncs modes, 15-15, 15-17
number of lines per frame, 15-35
number of samples, 15-34
operating modes, 15-4, 15-26
overview, 1-16
port width, 15-28
preamble, 15-7
programming model, 15-22
progressive video, 15-6
submodes for ITU-R 656, 15-9
and synchronization with DMA, 15-13
timer pins, 15-21
transfer delay, 15-19
TX modes with external frame syncs, 15-22
TX modes with internal frame syncs, 15-20
valid data detection, 15-15
vertical blanking interval only mode, 15-10
video frame partitioning, 15-7
video processing, 15-5
video streams, 15-8
when data transfer begins, 15-31
PPI_CLK cycle count, 15-34
PPI_CLK pin, 15-3
PPI_CLK signal, 15-26
PPI_CONTROL (PPI control) register, 15-26, 15-27
PPI control register (PPI_CONTROL), 15-26, 15-27
PPI_COUNT[15:0] field, 15-35
PPI_COUNT (transfer count) register, 15-34, 15-35
PPI_DELAY[15:0] field, 15-34
PPI_DELAY (delay count) register, 15-34
PPI_FRAME[15:0] field, 15-36
PPI_FRAME (lines per frame) register, 15-35, 15-36
PPI_FS1 signal, 15-26
PPI_FS2 signal, 15-26
PPI_FS3 signal, 15-33

PPI_STATUS (PPI status) register, 15-31, 15-32
preamble, PPI, 15-7
preboot, controlled by OTP programming, 17-4
preboot routine, 17-11
precharge all command, 7-34, 7-51
precharge command, 7-34, G-18
PREN bit, 14-22
prescale[6:0] field, 23-27
prescaler, RTC, 14-2
PRESCALE value, 23-4
priorities
  memory DMA operations, 6-48
  peripheral DMA operations, 6-48
prioritization
  DMA, 6-46 to 6-51
  interrupts, 5-4
private instructions, B-4
probes, oscilloscope, 19-11
processor
  dynamic power management, 18-1
  test features, B-1
processor block diagram, 1-7
programmable timing characteristics, EBIU, 7-11
program Pxn bit, 9-32
progressive video, 15-6
PROTOCOL_R (Rx protocol type) bits, 26-129
PROTOCOL_T (Tx protocol type) bits, 26-127
PR (promiscuous mode) bit, 21-61, 21-66
PS bit, 6-86
PSF (pass short frames) bit, 21-61, 21-65
PSM bit, 7-47, 7-56, 7-66, 7-70
PSSE bit, 7-44, 7-47, 7-56, 7-66, 7-70
PSSE (slave select enable) bit, 22-36, 22-37
public instructions, B-4
public JTAG scan instructions, B-5
PULSE_HI bit, 10-14, 10-16, 10-24, 10-41, 10-46
PULSE_HI toggle mode, 10-15
pulse width count and capture mode, *See* WDTH_CAP mode
pulse width modulation mode, *See* PWM_OUT mode
pulse width modulation mode, See PWM_OUT mode
pulse width modulator, 1-20
PUPSD bit, 7-66, 7-69
PWM_CLK clock, 10-20
PWM_CLK signal, 10-20
PWM_OUT mode, 10-10 to 10-23, 10-43
  control bit and register usage, 10-46
  error prevention, 10-44
  externally clocked, 10-20
  PULSE_HI toggle mode, 10-15
  stopping the timer, 10-21
Pxn bit, 9-30
Pxn both edges bit, 9-35
Pxn direction bit, 9-31
Pxn input enable bit, 9-32
Pxn polarity bits, 9-34
Pxn sensitivity bit, 9-35

Q

query semaphore, 19-3
quick boot, 17-41

R

RAF (receive all frames) bit, 21-61, 21-65
RBC bit, 6-38, 6-86
RBSY flag, 22-42
RBSY (receive error) bit, 22-40
RCKFE (clock falling edge select) bit, 24-34, 24-55, 24-58
RCVDATA16[15:0] field, 23-48
RCVDATA8[7:0] field, 23-47

RCVFLUSH (receive buffer flush) bit, 23-38, 23-39
RCVINTLEN (receive buffer interrupt length) bit, 23-38, 23-39
RCVSERVM (receive FIFO service interrupt mask) bit, 23-42
RCVSERV (receive FIFO service) bit, 23-42, 23-43
RCVSTAT[1:0] field, 23-40
RD_DLY (read strobe delay) bits, 20-16
RDIV
  field, 7-39, 7-55
  field, equation for value, 7-59
RDIV[11:0] field, 7-59
RD_RDY (read data ready) bit, 20-18
RDTYPE[1:0] field, 24-29, 24-55, 24-57
read
  asynchronous, 7-11
  command, 7-34
  transfers to SDRAM banks, 7-50
read access, for EBIU asynchronous memory controller, 7-19
read data ready (RD_RDY) bit, 20-18
read strobe delay (RD_DLY) bits, 20-16
read/write access bit, 3-7
read/write command, 7-49
READY (DMA ready) bit, 8-29
real-time clock, *See* RTC
real-time clock, See RTC
receive buffer[7:0] field, 25-27
receive configuration (SPORTx_RCR1, SPORTx_RCR2) registers, 24-54
receive data[15:0] field, 24-63
receive data[31:16] field, 24-63
receive data buffer[15:0] field, 22-44
received byte count in EP0 FIFO (EP0_RX_COUNT) bits, 26-126
receive FIFO, SPORT, 24-61
reception error, SPI, 22-42
refresh, SDRAM, 7-31
REGAD[4:0] field, 21-73
register-based DMA, 6-10
registers
  See also registers by name
  rotary counter, 13-19
  system, A-3
remote wakeup frame filters, MAC, 21-34
REP bit, 6-39, 6-86
REQPKT (request an IN transaction) bit, 26-111
REQPKT_RH (request an IN transaction) bit, 26-121
request and IN transaction (STALL_RECEIVED_RH) bit, 26-121
request and IN transaction (STALL_RECEIVED_TH) bit, 26-115
request an IN transaction (REQPKKT_RH) bit, 26-121
request an IN transaction (REQPKT) bit, 26-111
request an IN transaction (RXSTALL_TH) bit, 26-115
request data control command, DMA, 6-35
request data urgent control command, DMA, 6-35
RE (receive enable) bit, 21-42, 21-50, 21-61, 21-67
reserved SDRAM, 7-2
reset
  effect on SPI, 22-16
RESET_DOUBLE, 16-39, 17-100
RESET_DOUBLE bit, 16-39
reset endpoint data toggle (CLEAR_DATATOGGLE_R) bit, 26-121
reset endpoint data toggle (CLEAR_DATATOGGLE_T) bit, 26-115

RESET_OR_BABLE_BE (reset or bable IRQ enable) bit, 26-108
RESET_OR_BABLE_B (reset or bable indicator) bit, 26-107
reset or bable indicator (RESET_OR_BABLE_B) bit, 26-107
reset or bable IRQ enable (RESET_OR_BABLE_BE) bit, 26-108
RESET pin, 17-6
resets
  core and system, 17-138
  core double-fault, 17-6
  core-only software, 17-6
  hardware, 17-5, 17-8
  interrupts, 17-10
  software, 17-7
  system software, 17-5
  watchdog timer, 17-5, 17-7
RESET_SOFTWARE, 16-39, 17-100
RESET_SOFTWARE bit, 16-39
RESET (USB reset) bit, 26-98
reset vector, 17-1
RESET_WDOG, 16-39, 17-100
RESET_WDOG bit, 12-5, 16-39, 16-43
resource sharing, with semaphores, 19-2
restart control command, DMA, 6-33, 6-34
restart or finish control command, transmit, 6-35, 6-36
restrictions
  DMA control commands, 6-35
  DMA work unit, 6-25
RESUME_BE (resume signalling IRQ enable) bit, 26-108
RESUME_B (resume signalling indicator) bit, 26-107
resume mode flag (RESUME_MODE) bit, 26-98
RESUME_MODE (resume mode flag) bit, 26-98
resume signalling indicator (RESUME_B) bit, 26-107
resume signalling IRQ enable (RESUME_BE) bit, 26-108
RETI register, 17-10
RFS pins, 24-31
RFSR (receive frame sync required select) bit, 24-31, 24-32, 24-55, 24-57
RFS signal, 24-19
RFSx signal, 24-5
RLSBIT (receive bit order) bit, 24-55, 24-57
RMII
  multiplexing, 21-5
  pins, 21-6
RMII_IO (RMII port speed selector) bit, 21-61, 21-62
RMII (RMII mode) bit, 21-61, 21-63
ROM, 1-9, 7-1
rotary counter registers, 13-19
round robin operation, MDMA, 6-49
route Rx IRQ to INTx (RX_INTx_R) bits, 26-102
route Tx IRQ to INTx (TX_INTx_R) bits, 26-102
route USB/VBUS IRQ to INTx (USB_INTx_R) bits, 26-102
routing of interrupts, 5-16, 5-17
ROVF (sticky receive overflow status) bit, 24-63, 24-64, 24-65
row activation, SDRAM, 7-31
row address, 7-63
row precharge, SDRAM, 7-31
RPOLC (IrDA Rx polarity change) bit, 25-32, 25-33
RRFST (left/right order) bit, 24-13, 24-56, 24-58
RSCLKx pins, 24-30

RSCLKx signal, 24-5
RSFSE (receive stereo frame sync enable) bit, 24-11, 24-56, 24-58
RSPEN (receive enable) bit, 24-10, 24-54, 24-55, 24-56
RSTART (repeat start) bit, 23-32, 23-33
RSTC (reset all counters) bit, 21-43, 21-121, 21-122
RTC, 1-23, 14-1 to 14-27
  alarm clock features, 14-2, 14-25
  block diagram, 14-4
  clock rate, 14-5
  clock requirements, 14-5
  code examples, 14-22
  counters, 14-2
  deep sleep, 14-10
  digital watch features, 1-3, 14-1
  disabling prescaler, 14-6
  enabling prescaler, 14-5, 14-23
  event flags, 14-11
  initializing, 14-6
  interfaces, 14-3
  interrupt, 18-10
  interrupts, 14-7, 14-14
  interrupt structure, 14-16
  overview, 1-24
  prescaler, 14-2
  programming model, 14-7
  reads, 14-10
  registers, table, 14-19
  setting time of day, 14-13
  state transitions, 14-16
  stopwatch, 14-2, 14-13, 14-23
  synchronization, 14-6
  system state transition events, 14-17
  test mode, 14-6
  write latency, 14-9
  writes, 14-8, 14-9
RTC_ALARM (RTC alarm) register, 14-2, 14-19, 14-22
RTC_ICTL (RTC interrupt control) register, 14-19, 14-20
RTC_ISTAT (RTC interrupt status) register, 14-19, 14-21
RTC_PREN (prescaler enable) register prescaler enable (RTC_PREN) register, 14-5, 14-19
RTC_STAT (RTC status) register, 14-13, 14-19, 14-20
RTC status (RTC_STAT) register, 14-13, 14-19, 14-20
RTC_SWCNT (RTC stopwatch count) register, 14-2, 14-13, 14-19, 14-21
RUVF (sticky receive underflow status) bit, 24-63, 24-64, 24-65
RWKE (remote wakeup frame enable) bit, 21-35, 21-79, 21-80
RWKS[3:0] field, 21-79
RX_ACCEPT (received filtered frame interrupt enable) bit, 21-95, 21-100, 21-104
RX_ADDR (address filter failed) bit, 21-95, 21-98, 21-100, 21-102, 21-104
RX_ALIGN (alignment error) bit, 21-95, 21-98, 21-100, 21-103, 21-104
RX_ALIGN_CNT (MAC control frames received counter interrupt) bit, 21-113, 21-115
RX_ALLF_CNT (frames received all counter interrupt) bit, 21-113, 21-115
RX_BROAD (broadcast frames detected) bit, 21-95, 21-97, 21-100, 21-101, 21-104
RX_BROAD_CNT (broadcast frames received OK counter interrupt enable) bit, 21-113, 21-115
RXCKS (enable receive frame TCP/UDP checksum computation) bit, 21-90

RX_COMP (receive complete) bit, 21-95, 21-99, 21-100, 21-103, 21-104
RX_COUNT (USB Rx byte count) bits, 26-127
RX_CRC (frame CRC error) bit, 21-95, 21-98, 21-100, 21-103, 21-104
RX_CTL (control frame interrupt enable) bit, 21-95, 21-96, 21-100, 21-101, 21-104
RX DMA direction error detected, 21-41
RXDMAERR (RX DMA direction error status) bit, 21-92
RX_DMAO (DMA overruns detected) bit, 21-95, 21-97, 21-100, 21-102, 21-104
RXDWA (receive frame DMA word alignment) bit, 21-17, 21-50, 21-90, 21-91
RX_EQ64_CNT (frames length equal to 64 received counter interrupt) bit, 21-113, 21-115
RX_FCS_CNT (frame check sequence errors counter interrupt) bit, 21-113, 21-115
RX_FRAG (frame fragments detected) bit, 21-95, 21-98, 21-100, 21-102, 21-104
RX frame status interrupt, 21-40
RX_FRLEN[10:0] field, 21-95, 21-99
RXFSINT (RX frame-status interrupt status) bit, 21-92, 21-93
RX_GE1024_CNT (frames length 1024-max received counter interrupt enable) bit, 21-113, 21-115
RX hold register, 24-62
RX_INTx_R (route Rx IRQ to INTx) bits, 26-102
RX_IRL_CNT (in-range length errors counter interrupt) bit, 21-113, 21-115
RX_LATE (late collisions detected) bit, 21-95, 21-97, 21-100, 21-102, 21-104
RX_LEN (length errors detected) bit, 21-95, 21-98, 21-100, 21-102, 21-104
RX_LONG_CNT (frame too long errors counter interrupt) bit, 21-113, 21-115
RX_LONG (frame too long) bit, 21-95, 21-99, 21-100, 21-103, 21-104
RX_LT1024_CNT (frames length 512-1023 received counter interrupt enable) bit, 21-113, 21-115
RX_LT128_CNT (frames length 65-127 received counter interrupt enable) bit, 21-113, 21-115
RX_LT256_CNT (frames length 128-255 received counter interrupt enable) bit, 21-113, 21-115
RX_LT512_CNT (frames length 256-511 received counter interrupt enable) bit, 21-113, 21-115
RX_MACCTL_CNT (MAC control frames received counter interrupt enable) bit, 21-115
RX modes with external frame syncs, 15-21
RX_MULTI_CNT (multicast frames received OK counter interrupt) bit, 21-113, 21-115
RX_MULTI field, 21-95
RX_MULTI (multicast frame interrupt enable) bit, 21-97, 21-100, 21-101, 21-104
RXNE (receive FIFO not empty status) bit, 24-65
RX_OCTET_CNT (octets received OK counter interrupt enable) bit, 21-113, 21-115

RX_OK_CNT (frames received OK counter interrupt enable) bit, 21-113, 21-115
RX_OK (good received frame interrupt enable) bit, 21-95, 21-99, 21-100, 21-103, 21-104
RX_OPCODE_CNT (unsupported opcodes received counter interrupt enable) bit, 21-113, 21-115
RX_ORL_CNT (out-of-range length field counter interrupt) bit, 21-113, 21-115
Rx packet serviced (SERVICED_RXPKTRDY) bit, 26-111
RX_PAUSE_CNT (PAUSE MAC control frames received counter interrupt enable) bit, 21-113, 21-115
RX_PHY (PHY error interrupt enable) bit, 21-95, 21-97, 21-100, 21-102, 21-104
RxPktRdy autoclear enable (AUTOCLEAR_R) bit, 26-121
RXPKTRDY (data packet receive indicator) bit, 26-111
RXPKTRDY_R (data packet in FIFO indicator) bit, 26-121
RX_POLL_INTERVAL (USB Rx poll interval) bits, 26-130
Rx protocol type (PROTOCOL_R) bits, 26-129
RX_RANGE (out-of-range length fields detected) bit, 21-95, 21-97, 21-100, 21-102, 21-104
RXREQ signal, 25-7
RXSE (RxSEC enable) bit, 24-56, 24-58
RX_SHORT_CNT (frames length less than 64 received counter interrupt) bit, 21-113, 21-115
RXS (RX data buffer status) bit, 22-23, 22-40
RXSTALL_TH (request an IN transaction) bit, 26-115
RX_TYPED_CNT (typed frames received counter interrupt) bit, 21-113, 21-115
RX_TYPE (typed frame interrupt enable) bit, 21-95, 21-96, 21-100, 21-101, 21-104
RX_UCTL (unsupported control frame) bit, 21-95, 21-96, 21-100, 21-101, 21-104
RX_UNI_CNT (unicast frames received OK counter interrupt enable) bit, 21-113, 21-115
RX_VLAN1 (VLAN1 frame interrupt enable) bit, 21-95, 21-96, 21-100, 21-101, 21-104
RX_VLAN2 (VLAN2 frames detected) bit, 21-95, 21-96, 21-100, 21-101, 21-104

S

SA10 pin, 7-54
SADDR[6:0] field, 23-30
SAMPLE/PRELOAD instruction, B-6
sampling clock period, UART, 25-8
sampling edge, SPORT, 24-34
SAV codes, 15-33
SAV signal, 15-5
SB (set break) bit, 25-22
scale value[7:0] field, 11-6
scaling, of core timer, 11-7
scan paths, B-5
SCCB bit, 23-27
scheduling, memory DMA, 6-48
SCKE pin, 7-52
SCK signal, 22-4, 22-12, 22-15, 22-16

SCLK, 2-4, 18-5
  changing frequency, 7-45
  derivation, 18-1
  EBIU, 7-1
  status by operating mode (table), 18-8
SCLOVR (serial clock override) bit, 23-32
SCL pin, 23-5
SCLSEN (serial clock sense) bit, 23-35, 23-36
SCL serial clock, 23-27
SCL (serial clock) signal, 23-4
SCOMPM (slave transfer complete interrupt mask) bit, 23-42
SCOMP (slave transfer complete) bit, 23-42, 23-44
scratch[7:0] field, 25-32
scratchpad memory, and booting, 17-22
SCTLE bit, 7-52, 7-53, 7-56, 7-65, 7-66, 7-67, 7-71
SDAOVR (serial data override) bit, 23-32
SDA pin, 23-5
SDASEN (serial data sense) bit, 23-35, 23-36
SDA (serial data) signal, 23-4
SDC, 1-12, 7-4, 7-24
  address mapping, 7-26
  address muxing, 7-41
  architecture, 7-39
  code examples, 7-76
  commands, 7-46
  component configurations, 7-25
  core and DMA arbitration, 7-43
  core transfers to SDRAM, 7-76
  disabled CLKOUT, 7-77
  features, 7-24
  initialization, 7-79
  no burst mode, 7-40
  operation, 7-38
  pin state during commands, 7-46
  power-up latency, 7-55
SDC, 1-12, 7-4, 7-24 *(continued)*
  registers, 7-59
  SA10 pin, 7-54
  self-refresh mode, 7-77
  transfers to SDRAM using byte mask, 7-77
SDC commands, 7-46
SDCI bit, 7-75
SDEASE bit, 7-75, 7-76
SDIR (slave transfer direction) bit, 23-30, 23-31
SDPUA bit, 7-75, 7-76
SDQM[1:0] encodings during writes, 7-50
SDRAM, 1-9
  A[10] pin, 7-54
  address connections, 7-41
  address mapping, 7-26
  auto-refresh command, 7-34, 7-51
  bank activate command, 7-33
  banks, 3-5
  bank size, 7-2
  burst length, 7-32
  burst type, 7-32
  CAS latency, 7-33
  clock enables, setting, 7-65
  column read/write, 7-31
  commands, 7-33
  configuration, 7-24
  ERMS command, 7-33
  exiting self-refresh mode, 7-52
  external bank, 7-32
  external memory, 7-26
  initialization, 7-47, 7-70
  interface controller commands, 7-46
  interface management, 7-46
  interface signals, 7-29
  internal banks, 7-27
  internal memory banks, 7-32

SDRAM, 1-9 *(continued)*
latency, 7-70
memory banks, 7-3
memory size, 7-32
memory space, 7-2
MRS command, 7-33
multibank operation, 7-42
NOP command, 7-53
operation parameters, initializing, 7-47
parallel connection, 7-28
performance, 7-30
power-up sequence, 7-44, 7-47, 7-70
precharge all command, 7-34
precharge command, 7-34
read command, 7-34
read transfers, 7-50
read/write command, 7-49
refresh, 7-31
reserved, 7-2
row activation, 7-31
row precharge, 7-31
self-refresh mode, 7-34, 7-52
sharing external, 7-69
size configuration, 7-61
sizes supported, 3-5, 7-24
smaller than 16M byte, 7-63
specification of system, 7-28
stall cycles, 7-30
start address, 7-2
system block diagram, 7-57, 7-58
timing specs, 7-35
write command, 7-34
SDRAM controller, *See* SDC
SDRAM control status register (EBIU_SDSTAT), 7-74
SDRAM memory, 3-1
SDRAM memory bank control (EBIU_SDBCTL) register, 7-61, 7-62
SDRAM memory global control (EBIU_SDGCTL) register, 7-65, 7-66
SDRAM refresh rate control (EBIU_SDRRC) register, 7-59
SDRS bit, 7-55, 7-75, 7-76
SDSRA bit, 7-75
seconds (1 Hz) event flag bit, 14-21
seconds (1 Hz) interrupt enable bit, 14-20
seconds[5:0] field, 14-20, 14-22
SELECTED_ENDPOINT (USB endpoint index) bits, 26-101, 26-109
self-refresh command, 7-52
self-refresh mode, 7-34, 7-52, 7-72, 7-77
entering, 7-52
exiting, 7-52
semaphores, 19-2
coherency, 19-3
example code, 19-3
query, 19-3
send setup token (SETUPPKT) bit, 26-111
send STALL handshake (SENDSTALL) bit, 26-111
send STALL handshake (STALL_SEND_R) bit, 26-121
send STALL handshake (STALL_SEND_T) bit, 26-115
SENDSTALL (send STALL handshake) bit, 26-111
SEN (slave enable) bit, 23-28, 23-29
serial
clock frequency, 22-34
communications, 25-5
data transfer, 24-4
scan paths, B-4
serial clock divide modulus[15:0] field, 24-65, 24-66
serial peripheral interface, *See* SPI
serial scan paths, B-5

SERRM (slave transfer error interrupt mask) bit, 23-42
SERR (slave transfer error) bit, 23-42, 23-44
SERVICED_RXPKTRDY (Rx packet serviced) bit, 26-111
SERVICED_SETUPEND (setup end serviced) bit, 26-111
session end/disconnect indicator (DISCON_B) bit, 26-107
session end/disconnect IRQ enable (DISCON_BE) bit, 26-108
session indicator (SESSION) bit, 26-132, 26-134
SESSION_REQ_BE (session request IRQ enable) bit, 26-108
SESSION_REQ_B (session request indicator) bit, 26-107
session request indicator (SESSION_REQ_B) bit, 26-107
session request IRQ enable (SESSION_REQ_BE) bit, 26-108
SESSION (session indicator) bit, 26-132, 26-134
set index[5:0] field, 3-7
SET_PHYAD macro, 21-127
set Pxn bit, 9-33
set Pxn interrupt A enable bit, 9-37
set Pxn interrupt B enable bit, 9-38
SET_REGAD macro, 21-127
setup
  for EBIU asynchronous memory controller, 7-19
  SDRAM clock enables, 7-65
setup end serviced (SERVICED_SETUPEND) bit, 26-111
SETUPEND (setup end) bit, 26-111
setup end (SETUPEND) bit, 26-111
SETUPPKT (send setup token) bit, 26-111
shared interrupts, 5-4
shorten startup counter chain (TM_SHORT_CHAIN) bit, 26-141
SIC, *See* system interrupt controller
SIC_IAR0 (system interrupt assignment 0) register, 5-11, 8-26, 8-29
SIC_IAR0 (system interrupt assignment register 0), 8-31
SIC_IMASK (system interrupt mask) register, 5-5
SIC registers, 5-10
signal integrity, 19-8
signalling, via semaphores., 19-2
sine wave input, 1-25
SingleCollisionFrames register, 21-57
single pulse generation, timer, 10-12
SINITM (slave transfer initiated interrupt mask) bit, 23-42
SINIT (slave transfer initiated) bit, 23-42, 23-44
size of accesses, timer registers, 10-35
SIZE (size of words) bit, 22-36, 22-37
SKIP_EN (skip enable) bit, 15-26, 15-27
SKIP_EO (skip even odd) bit, 15-27, 15-28
slave mode setup, in TWI, 23-11, 23-53
slaves
  EBIU, 7-5
  PAB, 2-6
slave select, SPI, 22-39
slave SPI device, 22-5
sleep mode, 1-26, 18-9
  MAC operation in, 21-32
SLEN[4:0] field, 24-50, 24-51, 24-56, 24-57
  restrictions, 24-28
  word length formula, 24-28
small descriptor mode, DMA, 6-15
small model mode, DMA, 6-70

SMODE_B (switch charge pump mode) bits, 26-141
SOF_BE (start of frame IRQ enable) bit, 26-108
SOF_B (start of frame indicator) bit, 26-107
soft connect enable (SOFTC_CONN) bit, 26-98
SOFT_CONN (soft connect enable) bit, 26-98
software
  interrupts, 5-3
  management of DMA, 6-51
  watchdog timer, 1-24, 12-1
software reset, 17-7, 17-99
software reset register (SWRST), 16-39, 17-100
source channels, memory DMA, 6-7
SOVFM (slave overflow interrupt mask) bit, 23-42
SOVF (slave overflow) bit, 23-42, 23-44
speculative read, and MAC, 21-43
SPE (SPI enable) bit, 22-36, 22-37
SPI, 22-2 to 22-53
  beginning and ending transfers, 22-22
  block diagram, 22-4
  clock phase, 22-12, 22-14, 22-17
  clock polarity, 22-12, 22-16
  clock signal, 22-4, 22-16
  code examples, 22-45
  compatible peripherals, 22-3
  data corruption, avoiding, 22-15
  data interrupt, 22-17
  data transfer, 22-15
  detecting transfer complete, 22-40
  and DMA, 22-11
  DMA initialization, 22-48
  DMA transfers, 22-48
  effect of reset, 22-16
  enabling the SPI system, 22-35
SPI, 22-2 to 22-53 *(continued)*
  error interrupt, 22-17
  error signals, 22-40 to 22-42
  features, 22-2
  full-duplex synchronous serial interface, 22-2
  general operation, 22-15 to 22-22
  initialization, 22-45
  internal interfaces, 22-11
  interrupt outputs, 22-17
  interrupts, 22-47
  master mode, 22-15, 22-18
  master mode DMA operation, 22-24
  mode fault error, 22-41
  multiple slave systems, 22-8
  overview, 1-20
  reception error, 22-42
  registers, table, 22-34
  SCK signal, 22-4
  slave boot mode, 17-69
  slave device, 22-5
  slave mode, 22-16, 22-20
  slave mode DMA operation, 22-27
  slave-select function, 22-38
  slave transfer preparation, 22-22
  SPI_FLG mapping to port pins, 22-39
  starting DMA transfer, 22-51
  starting transfer, 22-46
  stopping, 22-48
  stopping DMA transfers, 22-51
  switching between transmit and receive, 22-23
  timing, 22-6
  transfer formats, 22-12 to 22-14
  transfer initiate command, 22-18, 22-19
  transfer modes, 22-19
  transfer protocol, 22-14
  transmission error, 22-42
  transmission/reception errors, 22-40
  transmit collision error, 22-42

SPI, 22-2 to 22-53 *(continued)*
using DMA, 22-11
word length, 22-36
SPI_BAUD (SPI baud rate) register, 22-34
SPI_BAUD values, 22-35
SPI_CTL (SPI control) register, 22-5, 22-34, 22-35, 22-37
SPI_FLG (SPI flag) register, 22-7, 22-8, 22-34, 22-38
SPIF (SPI finished) bit, 22-10, 22-23, 22-40
SPI_RDBR shadow[15:0] field, 22-44
SPI RDBR shadow (SPI_SHADOW register), 22-34
SPI RDBR shadow (SPI_SHADOW) register, 22-44
SPI_RDBR (SPI receive data buffer) register, 22-34, 22-43, 22-44
SPI_SHADOW (SPI RDBR shadow) register, 22-34, 22-44
SPI slave select, 22-39
$\overline{\text{SPISS}}$ signal, 22-6, 22-8, 22-12
SPI_STAT (SPI status) register, 22-34, 22-40
SPI_TDBR (SPI transmit data buffer) register, 22-34, 22-42, 22-43
SPORT, 1-18, 24-1 to 24-78
2X clock recovery control, 24-25
active low vs. active high frame syncs, 24-34
channels, 24-15
clock, 24-30
clock frequency, 24-26, 24-65
clock rate, 24-2
clock rate restrictions, 24-27
companding, 24-30
configuration, 24-11
data formats, 24-29
data word formats, 24-58
SPORT, 1-18, 24-1 to 24-78 *(continued)*
disabling, 24-11
DMA data packing, 24-24
enable/disable, 24-10
enabling multichannel mode, 24-18
framed serial transfers, 24-32
framed vs. unframed, 24-31
frame sync, 24-33, 24-36
frame sync frequencies, 24-26
framing signals, 24-31
general operation, 24-10
H.100 standard protocol, 24-25
initialization code, 24-57
internal memory access, 24-39
internal vs. external frame syncs, 24-33
late frame sync, 24-18
modes, 24-11
moving data to memory, 24-39
multichannel frame, 24-20
multichannel operation, 24-15 to 24-25
multichannel transfer timing, 24-17
overview, 1-18
packing data, multichannel DMA, 24-24
peripheral access bus error, 24-40
pin/line terminations, 24-9
port connection, 24-7
receive and transmit functions, 24-4
receive clock signal, 24-30
receive FIFO, 24-61
receive word length, 24-62
register writes, 24-47
RX hold register, 24-62
sampling edge, 24-34
selecting bit order, 24-28
serial data communication protocols, 24-2
shortened active pulses, 24-11
signals, 24-5
single clock for both receive and transmit, 24-30

SPORT, 1-18, 24-1 to 24-78 *(continued)*
single word transfers, 24-39
stereo serial connection, 24-9
stereo serial frame sync modes, 24-18
stereo serial operation, 24-11
support for standard protocols, 24-25
termination, 24-9
throughput, 24-7
timing, 24-40
transmit clock signal, 24-30
transmitter FIFO, 24-59
transmit word length, 24-59
TX hold register, 24-59
TX interrupt, 24-60
unframed data flow, 24-32
unpacking data, multichannel DMA, 24-24
window offset, 24-22
word length, 24-28
SPORT error interrupt, 24-39
SPORT registers, table, 24-46
SPORT RX interrupt, 24-39, 24-63
SPORT TX interrupt, 24-39
SPORTx_CHNL (SPORTx current channel) registers, 24-68
SPORTx_MCMCn (SPORTx multichannel configuration) registers, 24-67
SPORTx_MRCSn (SPORTx multichannel receive select) registers, 24-23, 24-24, 24-69
SPORTx_MTCSn (SPORTx multichannel transmit select) registers, 24-23, 24-24, 24-70
SPORTx_RCLKDIV (SPORTx receive serial clock divider) registers, 24-65
SPORTx_RCR1 (SPORTx receive configuration 1) registers, 24-54
SPORTx_RCR2 (SPORTx receive configuration 2) registers, 24-54, 24-56
SPORTx_RFSDIV (SPORTx receive frame sync divider) registers, 24-66
SPORTx_RX (SPORTx receive data) registers, 24-19, 24-61
SPORTx_STAT (SPORTx status) registers, 24-64
SPORTx_TCLKDIV (SPORTx transmit serial clock divider) registers, 24-65
SPORTx_TCR1 (transmit configuration 1) register, 24-48
SPORTx_TCR2 (transmit configuration 2) register, 24-48
SPORTx_TFSDIV (SPORTx transmit frame sync divider) registers, 24-66
SPORTx_TX (SPORTx transmit data) registers, 24-19, 24-38, 24-59
SRAM, 1-9, 7-1
interface, 19-5
SRAM ADDR[13:12] field, 3-7
SRFS bit, 7-52, 7-66, 7-67, 7-71
SSEL[3:0] field, 2-4, 18-5, 18-26
SSEL bit, 19-2
SSEL (system select) bit, 18-26
STABUSY (STA busy status) bit, 21-73, 21-74
STADATA[15:0] field, 21-74
STADISPRE (disable preamble generation) bit, 21-73
STAIE (station management transfer done interrupt enable) bit, 21-73
stall cycles, SDRAM, 7-30
STALL handshake received (STALL_RECEIVED) bit, 26-111
STALL handshake sent (STALL_SENT) bit, 26-111
STALL handshake sent (STALL_SENT_R) bit, 26-121

STALL handshake sent
(STALL_SENT_T) bit, 26-115
STALL_RECEIVED_RH (request and IN
transaction) bit, 26-121
STALL_RECEIVED (STALL handshake
received) bit, 26-111
STALL_RECEIVED_TH (request and IN
transaction) bit, 26-115
STALL_SEND_R (send STALL
handshake) bit, 26-121
STALL_SEND_T (send STALL
handshake) bit, 26-115
STALL_SENT_R (STALL handshake
sent) bit, 26-121
STALL_SENT (STALL handshake sent)
bit, 26-111
STALL_SENT_T (STALL handshake
sent) bit, 26-115
STAOP (station management operation
code) bit, 21-73
start address registers
(DMAx_START_ADDR), 6-76
(MDMA_yy_START_ADDR), 6-76
start of frame indicator (SOF_B) bit,
26-107
start of frame IRQ enable (SOF_B) bit,
26-108
state transitions, RTC, 14-16
station management
read transfer, MAC, 21-49
transfer done, 21-41
write transfer, MAC, 21-48
STATUS[1:0] field, 25-29, 25-30
status (CNT_STATUS) register, 13-19,
13-22
STATUSPKT_H (packet transaction
status) bit, 26-111
STB (stop bits) bit, 25-22
STDVAL (slave transmit data valid) bit,
23-28, 23-29
stereo serial
data, 24-3
device, SPORT connection, 24-9
frame sync modes, 24-18
operation, SPORT, 24-11
STMDONE (station management transfer
done interrupt status) bit, 21-92
STOPCK (stop clock) bit, 18-26, 18-27
stop clock (STOPCK) bit, 18-26, 18-27
STOP (issue stop condition) bit, 23-33
stop mode, DMA, 6-11, 6-69
stopping DMA transfers, 6-29
stopwatch, 14-13
stopwatch count[15:0] field, 14-21
stopwatch event flag bit, 14-21
stopwatch function, RTC, 14-2
stopwatch interrupt enable bit, 14-20
STP (stick parity) bit, 25-22
streams, memory DMA, 6-7
subbank access[1:0] field, 3-7
subbanks
L1 data memory, 3-4
L1 instruction memory, 3-3
supervisor mode, 17-10
surface-mount capacitors, 19-9
SUSPEND_BE (suspend signalling IRQ
enable) bit, 26-108
SUSPEND_B (suspend signalling
indicator) bit, 26-107
suspend mode enable
(SUSPEND_MODE) bit, 26-98
suspend mode output enable
(ENABLE_SUSPENDM) bit, 26-98
SUSPEND_MODE (suspend mode
enable) bit, 26-98
suspend signalling indicator
(SUSPEND_B) bit, 26-107
suspend signalling IRQ enable
(SUSPEND_BE) bit, 26-108

switch charge pump mode (SMODE_B) bits, 26-141
SWRESET, 16-47, 16-48, 17-102
SWRST, software reset register, 17-99
SWRST (software reset register), 16-39, 17-100
SYNC bit, 6-25, 6-26, 6-27, 6-63, 6-69, 6-71, 25-18
synchronization
  interrupt-based methods, 6-52
  of descriptor queue, 6-58
  of DMA, 6-51 to 6-62
synchronized transition, DMA, 6-28
synchronous serial data transfer, 24-4
synchronous serial ports, *See* SPORT
SYSCR (System Reset Configuration Register), 17-102
SYSCR (system reset configuration register), 16-47, 16-48, 16-53, 16-56, 17-100, 17-102
system
  interrupt controller, 5-2, 10-7
  interrupt processing, 5-8
  interrupts, 5-1, 5-2
  peripherals, 1-5
system and core event mapping (table), 5-3
system clock, 1-25
system clock, changing during runtime, 7-44
system clock (SCLK), 18-1
  managing, 19-2
system design, 19-1 to 19-12
  high frequency considerations, 19-7
  recommendations and suggestions, 19-9
  recommended reading, 19-11
system interrupt assignment 0 (SIC_IAR0) register, 5-11
system interrupt assignment register 0 (SIC_IAR0), 8-26, 8-29, 8-31
system interrupt controller (SIC), 5-2
  controlling interrupts, 5-4
  enabling flexible interrupt handling, 10-7
  enabling individual peripheral interrupts, 5-4
  main functions of, 5-4
  peripheral interrupt events, 5-18
  registers, 5-10
system interrupt mask (SIC_IMASK) register, 5-5
system peripheral clock, *See* SCLK
SYSTEM_RESET, 16-39, 17-100
SYSTEM_RESET[2:0] field, 16-39, 16-43
System Reset Configuration Register (SYSCR), 17-102
system reset configuration register (SYSCR), 16-47, 16-48, 16-53, 16-56, 17-100, 17-102
system select (SSEL) bit, 18-26
system software reset, 17-5
SZ (send zero) bit, 22-21, 22-37

T

TAP registers
  boundary-scan, B-7
  BYPASS, B-6
  instruction, B-2, B-4
TAP (test access port), B-1, B-2
  controller, B-2
target address, 17-29
TARGET_EP_NO_R (target EPx number) bits, 26-129
TARGET_EP_NO_T (target EPx number) bits, 26-127
target EPx number (TARGET_EP_NO_R) bits, 26-129
target EPx number (TARGET_EP_NO_T) bits, 26-127
TAUTORLD bit, 11-3, 11-5

TCKFE (clock drive/sample edge select) bit, 24-34, 24-49, 24-53
TCNTL (core timer control) register, 11-3, 11-5
TCOUNT (core timer count) register, 11-3, 11-5
TCP/IP-style checksums, MAC, 21-21
TCSR bit, 7-66, 7-74
TDM interfaces, 24-4
TDTYPE[1:0] field, 24-29, 24-49, 24-51
TEMT (TSR and UARTx_THR empty) bit, 25-7, 25-25, 25-26
termination, DMA, 6-29
terminations, SPORT pin/line, 24-9
test access port (TAP), B-1, B-2
  controller, B-2
test clock (TCK), B-6
test features, B-1 to B-7
testing circuit boards, B-1, B-6
test-logic-reset state, B-4
test point access, 19-11
TESTSET instruction, 2-14, 19-3
TE (transmitter enable) bit, 21-43, 21-51, 21-61, 21-65
TFS pins, 24-31, 24-38
TFSR (transmit frame sync required select) bit, 24-31, 24-32, 24-49, 24-52
TFS signal, 24-19
TFSx signal, 24-5
THRE flag, 25-6, 25-17
THRE (THR empty) bit, 25-12, 25-25, 25-26
throughput
  DAB, 2-14
  DMA, 6-43
  from DMA system, 6-42
  general-purpose ports, 9-11
  SPORT, 24-7
TIMDISx bit, 10-36, 10-37
time-division-multiplexed (TDM) mode, 24-15
  See also SPORT, multichannel operation
TIMENx bit, 10-35, 10-36
timeout error (ERROR_H) bit, 26-111
timeout error indicator (ERROR_RH) bit, 26-121
timeout error indicator (ERROR_TH) bit, 26-115
TIMEOUT (host timeout) bit, 8-29
timer configuration (TIMERx_CONFIG) registers, 10-5, 10-40, 10-41
timer counter[15:0] field, 10-43
timer counter[31:16] field, 10-43
timer counter (TIMERx_COUNTER) registers, 10-4, 10-41, 10-43
TIMER_DISABLE bit, 10-46
TIMER_DISABLE (timer disable) register, 10-5, 10-37
timer disable (TIMER_DISABLE) register, 10-5, 10-37
TIMER_ENABLE bit, 10-46
TIMER_ENABLE (timer enable) register, 10-5, 10-35, 10-36, 15-24
timer enable (TIMER_ENABLE) register, 10-5, 10-35, 10-36
timer input select (TIN_SEL) bit, 10-41, 10-46
timer interrupt (TIMILx) bits, 10-4, 10-39
timer period[15:0] field, 10-45
timer period[31:16] field, 10-45
timer period (TIMERx_PERIOD) registers, 10-4, 10-43, 10-45
timers, 10-1 to 10-57
  core, 11-1 to 11-8
  EXT_CLK mode, 10-31 to 10-33
  overview, 1-20
  watchdog, 1-24, 12-1 to 12-10
  WDTH_CAP mode, 25-16

TIMER_STATUS (timer status) register, 10-5, 10-37, 10-39
timer status (TIMER_STATUS) register, 10-5, 10-37, 10-39
timer width[15:0] field, 10-45
timer width[31:16] field, 10-45
timer width (TIMERx_WIDTH) registers, 10-43, 10-45
TIMERx_CONFIG (timer configuration) registers, 10-5, 10-40, 10-41
TIMERx_COUNTER (timer counter) registers, 10-4, 10-41, 10-43
TIMERx_PERIOD (timer period) registers, 10-4, 10-43, 10-45
TIMERx_WIDTH (timer width) registers, 10-43, 10-45
TIMILx (timer interrupt) bits, 10-4, 10-39
timing
  auto-refresh, 7-59
  memory DMA, 6-45
  multichannel transfer, 24-17
  peripherals, 2-4
  SPI, 22-6
timing examples, for SPORTs, 24-40
timing specs, SDRAM, 7-35
TIMOD[1:0] field, 22-17, 22-19, 22-36, 22-37
TIN_SEL (timer input select) bit, 10-41, 10-46
TINT bit, 11-3, 11-5
TLSBIT (bit order select) bit, 24-49, 24-51
TMODE[1:0] field, 10-11, 10-41, 10-46
TM_PLL_VCO (boost PLL amplitude) bit, 26-141
TMPWR bit, 11-3, 11-5
TMRCLK input, 10-58
TMREN bit, 11-3, 11-5
TMR pin, 10-47
TMRx pins, 10-3, 10-15
TM_SELC (increase PLL charge pump current) bit, 26-141
TM_SHORT_CHAIN (shorten startup counter chain) bit, 26-141
TOGGLE_HI bit, 10-41, 10-47
TOGGLE_HI mode, 10-16
toggle Pxn bit, 9-34
toggle Pxn interrupt A enable bit, 9-41
toggle Pxn interrupt B enable bit, 9-42
tools, development, 1-29
TOVF_ERRx bit, 10-25, 10-28
TOVF_ERRx bits, 10-4, 10-7, 10-15, 10-39, 10-40, 10-48
TOVF (transmit overflow status) bit, 24-60, 24-64, 24-65
TPOLC (IrDA TX polarity change) bit, 25-32, 25-33
traffic control, DMA, 6-46 to 6-51
transfer count (ECCCNT) bits, 20-21
transfer count (PPI_COUNT) register, 15-34, 15-35
transfer frame protocol, MAC, 21-9
transfer initiate command, 22-18, 22-19
transfer initiation from SPI master, 22-19
transfer rate
  memory DMA channels, 6-43
  peripheral DMA channels, 6-43
transfers
  memory-to-memory, 6-7
transfer size (TxferSize), 26-32, 26-35
transitions
  continuous DMA, 6-25
  DMA work unit, 6-25
  operating mode, 18-11, 18-12
  synchronized DMA, 6-25
transmission error, SPI, 22-42
transmit clock, serial (TSCLKx) pins, 24-30
transmit collision error, SPI, 22-42

transmit configuration registers (SPORTx_TCR1 and SPORTx_TCR2), 24-48
transmit data[15:0] field, 24-61
transmit data[31:16] field, 24-61
transmit data buffer[15:0] field, 22-43
transmit hold[7:0] field, 25-27
t_{RAS}, 7-35
TRAS[3:0] field, 7-66, 7-68
t_{RC}, 7-37
t_{RCD}, 7-36
TRCD[2:0] field, 7-66, 7-68
t_{REF}, 7-37
t_{REFI}, 7-38
t_{RFC}, 7-37
TRFST (left/right order) bit, 24-50, 24-53
triggering DMA transfers, 6-62
t_{RP}, 7-36
TRP[2:0] field, 7-66, 7-68
t_{RRD}, 7-36
TRUNx bits, 10-21, 10-37, 10-39, 10-47
TSCALE (core timer scale) register, 11-3, 11-7
TSCLKx signal, 24-5
TSFSE (transmit stereo frame sync enable) bit, 24-10, 24-11, 24-50, 24-53
TSPEN (transmit enable) bit, 24-48, 24-49, 24-50
tuning of DPHY clocks (CNOS) bits, 26-140
TUVF (transmit underflow status) bit, 24-38, 24-60, 24-64, 24-65
TWI, 1-15, 23-2 to 23-60
  block diagram, 23-3
  bus arbitration, 23-8
  clock generation, 23-7
  controller, 23-2
  electrical specifications, 23-60
TWI, 1-15, 23-2 to 23-60 *(continued)*
  fast mode, 23-10
  features, 23-2
  general call address, 23-10
  general setup, 23-11
  I^2C compatibility, 1-15
  master boot mode, 17-72
  master mode clock setup, 23-12
  master mode receive, 23-14
  master mode transmit, 23-13
  peripheral interface, 23-5
  pins, 23-5
  slave boot mode, 17-75
  slave mode operation, 23-11
  start and stop conditions, 23-9
  synchronization, 23-7
  transfer protocol, 23-6
TWI_CLKDIV (SCL clock divider) register, 23-27, 23-28
TWI_CONTROL (TWI control) register, 23-4, 23-27
TWI_ENA bit, 23-27
TWI_FIFO_CTL (TWI FIFO control) register, 23-38
TWI_FIFO_STAT (TWI FIFO status) register, 23-40
TWI_INT_STAT (TWI interrupt status) register, 23-42
TWI_MASTER_CTL (TWI master mode control) register, 23-32
TWI_MASTER_STAT (TWI master mode status) register, 23-35
TWI_SLAVE_ADDR (TWI slave mode address) register, 23-30
TWI_SLAVE_CTL (TWI slave mode control) register, 23-28
TWI_SLAVE_STAT (TWI slave mode status) register, 23-30
two-dimensional DMA, 6-12
two-wire interface, *See* TWI

t_{WR}, 7-36
TWR[1:0] field, 7-66, 7-69
TX_ABORTC_CNT (frames aborted due to excess collisions counter interrupt enable) bit, 21-117, 21-119
TX_ABORT_CNT (transmission aborted frames counter interrupt) bit, 21-117, 21-119
TX_ALLF_CNT (frames transmitted all counter interrupt) bit, 21-117, 21-119
TX_BROAD_CNT (broadcast frames transmitted OK counter interrupt enable) bit, 21-117, 21-119
TX_BROAD (Tx broadcast frames detected) bit, 21-107, 21-109, 21-110, 21-112
TX_CCNT[3:0] field, 21-105, 21-106
TXCOL flag, 22-42
TXCOL (transmit collision error) bit, 22-40
TX_COMP (frame transmissions complete) bit, 21-105, 21-108, 21-109, 21-111
TX_COUNT (USB Tx byte count) bits, 26-131
TX_CRS_CNT (carrier sense errors counter interrupt enable) bit, 21-117, 21-119
TX_CRS (no carrier) bit, 21-105, 21-106, 21-109, 21-110, 21-112
TX_DEFER_CNT (frames with deferred transmission counter interrupt) bit, 21-117, 21-119
TX_DEFER (frame deferral interrupt enable) bit, 21-105, 21-106, 21-109, 21-110, 21-112
TX DMA direction error detected, 21-41
TXDMAERR (TX DMA direction error status) bit, 21-29, 21-92
TX_DMAU (DMA underrun) bit, 21-105, 21-107
TXDWA (transmit frame DMA word alignment) bit, 21-27, 21-90
TX_ECOLL (excessive collision errors detected) bit, 21-105, 21-107, 21-109, 21-111, 21-112
TX_EDEFER (excessive deferrals detected) bit, 21-105, 21-107, 21-109, 21-110, 21-112
TX_EQ64_CNT (frames length equal to 64 transmitted counter interrupt enable) bit, 21-117, 21-119
TX_ER pin, 21-5
TXE (transmission error) bit, 22-40, 22-42, 24-60, 24-65
TX_EXDEF_CNT (frames with excessive deferral counter interrupt enable) bit, 21-117, 21-119
TxferSize (transfer size), 26-32, 26-35
TX frame status interrupt, 21-40
TX_FRLEN[10:0] field, 21-105
TXFSINT (TX frame-status interrupt status) bit, 21-92, 21-93
TXF (transmit FIFO full status) bit, 24-64
TX_GE1024_CNT (frames length 1024-max transmitted counter interrupt) bit, 21-117, 21-119
TX hold register, 24-59
TXHRE (transmit hold register empty) bit, 24-65
TX_INTx_R (route Tx IRQ to INTx) bits, 26-102
TX_LATE_CNT (late collisions counter interrupt enable) bit, 21-117, 21-119
TX_LATE (late collision error) bit, 21-105, 21-107, 21-109, 21-110, 21-112
TX_LOSS (losses of carrier detected) bit, 21-105, 21-106, 21-109, 21-112

TX_LOST_CNT (frames lost due to internal MAC transmit error counter interrupt enable) bit, 21-117, 21-119
TX_LT1024_CNT (frames length 512-1023 transmitted counter interrupt enable) bit, 21-117, 21-119
TX_LT128_CNT (frames length 65-127 transmitted counter interrupt) bit, 21-117, 21-119
TX_LT256_CNT (frames length 128-255 transmitted counter interrupt) bit, 21-117, 21-119
TX_LT512_CNT (frames length 256-511 transmitted counter interrupt) bit, 21-117, 21-119
TX_MACCTL_CNT (MAC control frames transmitted counter interrupt) bit, 21-117, 21-119
TX_MACE (internal MAC errors detected) bit, 21-109, 21-110, 21-112
TX_MCOLL_CNT (multiple collision frames counter interrupt enable) bit, 21-117, 21-119
TX multicast, TX broadcast[1:0] field, 21-105
TX_MULTI_CNT (multicast frames transmitted OK counter interrupt enable) bit, 21-117, 21-119
TX_MULTI (multicast frames detected) bit, 21-107, 21-109, 21-110, 21-112
TX_OCTET_CNT (octets transmitted OK counter interrupt enable) bit, 21-117, 21-119
TX_OK_CNT (frames transmitted OK counter interrupt enable) bit, 21-117, 21-119
TX_OK (frames transmitted OK) bit, 21-105, 21-108, 21-109, 21-111
TxPktRdy autoset enable (AUTOSET_T) bit, 26-115
TXPKTRDY (data packet in FIFO indicator) bit, 26-111
TXPKTRDY_T (data packet in FIFO indicator) bit, 26-115
TX_POLL_INTERVAL (USB Tx poll interval) bits, 26-128
Tx protocol type (PROTOCOL_T) bits, 26-127
TXREQ signal, 25-6
TX_RETRY (late collisions detected) bit, 21-105, 21-109, 21-112
TX_SCOLL_CNT (single collision frames counter interrupt enable) bit, 21-117, 21-119
TXSE (TxSEC enable) bit, 24-50, 24-53
t_{XSR}, 7-37
TXS (SPI_TDBR data buffer status) bit, 22-23, 22-40
TX_UNI_CNT (unicast frames transmitted OK counter interrupt) bit, 21-117, 21-119
type definitions, MAC, 21-123
TypedFramesReceived register, 21-56

U

UART, 1-21, 25-2 to 25-42
  assigning interrupt priority, 25-13
  autobaud detection, 25-14
  baud rate, 25-7
  baud rate examples, 25-14
  bit rate examples, 25-14
  bit rate generation, 25-13
  bitstream, 25-6
  block diagram, 25-3
  booting, 25-15
  character transmission, 25-37
  clearing interrupt latches, 25-30
  clock, 25-13
  clock rate, 2-4

UART, 1-21, 25-2 to 25-42 *(continued)*
code examples, 25-33
connected to peripheral access bus, 25-4
data communication via infrared signals, 25-5
data words, 25-5
divisor reset, 25-31
DMA channels, 25-18
DMA mode, 25-18
errors during reception, 25-8
external interfaces, 25-3
features, 25-2
glitch filtering, 25-10
initialization, 25-33
internal TSR register, 25-7
interrupt conditions, 25-29
interrupts, 25-11
IrDA mode, 25-2
IrDA receiver, 25-9
IrDA receiver pulse detection, 25-11
IrDA transmit pulse, 25-9
IrDA transmitter, 25-9
loopback mode, 25-24
mixing modes, 25-19
modem status, 25-4
non-DMA interrupt operation, 25-39
non-DMA mode, 25-16
polling, 25-38
receive operation, 25-7
receive sampling window, 25-10
registers, table, 25-20
sampling clock period, 25-8
standard, 25-2
string transmission, 25-37
switching from DMA to non-DMA, 25-19
switching from non-DMA to DMA, 25-20
and system DMA, 25-28
transmission, 25-6
transmission SYNC bit use, 25-40
UART ports
overview, 1-21
UART receive buffer registers (UARTx_RBR), 25-7
UARTx_DLH (UART divisor latch high byte) registers, 25-21, 25-31
UARTx_DLL (UART divisor latch low byte) registers, 25-20, 25-31
UARTx_GCTL (UART global control) registers, 25-21, 25-32
UARTx_IER (UART interrupt enable) registers, 25-21, 25-27, 25-28
UARTx_IIR (UART interrupt identification) registers, 25-13, 25-21, 25-30
UARTx_LCR (UART line control) registers, 25-6, 25-21, 25-22
UARTx_LSR (UART line status) registers, 25-21, 25-25
UARTx_MCR (UART modem control) registers, 25-21, 25-24
UARTx_RBR (UART receive buffer) registers, 25-7, 25-20, 25-27
UARTx_SCR (UART scratch) registers, 25-21, 25-32
UARTx_THR (UART transmit holding) registers, 25-6, 25-20, 25-27
UCEN (enable UART clocks) bit, 25-7, 25-13, 25-31, 25-32, 25-33
UCIE (up count interrupt enable) bit, 13-21
UCII (up count interrupt identifier) bit, 13-22
UNDERRUN_T (no TxPktRdy for IN token) bit, 26-115
UNDR (FIFO underrun) bit, 15-32, 15-33
unframed/framed, serial data, 24-31
UnicastFramesReceivedOK register, 21-54
UnicastFramesXmittedOK register, 21-58

universal asynchronous receiver/transmitter, *See* UART
unpopulated memory, 7-9
UnsupportedOpcodesReceived register, 21-56
unused pins, 19-13
Upper PBS01 Half Page (PBS01H, Bits 15–0), 17-113
Upper PBS01 Half Page (PBS01H, Bits 63–16), 17-112
urgency threshold enable (UTE) bit, 6-41
USB_APHY_CNTRL2 (USB APHY control 2) register, 26-140
USB APHY control 2 (USB_APHY_CNTRL2) register, 26-140
USB common interrupts enable (USB_INTRUSBE) register, 26-108
USB common interrupts (USB_INTRUSB) register, 26-107
USB control/status EP0 (USB_CSR0) register, 26-111
USB_COUNT0 (USB received byte count in EP0 FIFO) register, 26-126
USB_CSR0 (USB control/status EP0) register, 26-111
USB DMA endpoint x interrupt (DMAx_INT) bits, 26-143
USB_DMA_INTERRUPT (USB DMA interrupt) register, 26-143
USB DMA interrupt (USB_DMA_INTERRUPT) register, 26-143
USB DMAx address high (USB_DMAxADDRHIGH) register, 26-146
USB DMAx address low (USB_DMAxADDRLOW) register, 26-146
USB_DMAxADDRHIGH (USB DMAx address high) register, 26-146
USB_DMAxADDRLOW (USB DMAx address low) register, 26-146
USB_DMAxCONTROL (USB DMAx control) registers, 26-144
USB DMAx control (USB_DMAxCONTROL) registers, 26-144
USB_DMAxCOUNTHIGH (USB DMAx count high) register, 26-148
USB DMAx count high (USB_DMAxCOUNTHIGH) register, 26-148
USB_DMAxCOUNTLOW (USB DMAx count low) register, 26-147
USB DMAx count low (USB_DMAxCOUNTLOW) register, 26-147
USB enable (GLOBAL_ENA) bit, 26-96
USB endpoint index (SELECTED_ENDPOINT) bits, 26-101, 26-109
USB_FADDR (USB function address) register, 26-100
USB frame number (FRAME_NUMBER) bits, 26-109
USB frame number (USB_FRAME) register, 26-109
USB_FRAME (USB frame number) register, 26-109
USB_FS_EOF1 (USB full-speed EOF 1) register, 26-138
USB full-speed EOF 1 (USB_FS_EOF1) register, 26-138
USB function address (USB_FADDR) register, 26-100
USB global control (USB_GLOBAL_CTL) register, 26-96

USB_GLOBAL_CTL (USB global control) register, 26-96
USB global interrupt (USB_GLOBINTR) register, 26-102
USB_GLOBINTR (USB global interrupt) register, 26-102
USB hibernate signal (CSR_HBR) bit, 26-140
USB high-speed EOF 1 (USB_HS_EOF1) register, 26-138
USB_HS_EOF1 (USB high-speed EOF 1) register, 26-138
USB_INDEX (USB index) register, 26-101, 26-109
USB index (USB_INDEX) register, 26-101, 26-109
USB_INTRRXE (USB receive interrupt enable) register, 26-106
USB_INTRRX (USB receive interrupt) register, 26-104
USB_INTRTXE (USB transmit interrupt enable) register, 26-105
USB_INTRTX (USB transmit interrupt) register, 26-103
USB_INTRUSBE (USB common interrupts enable) register, 26-108
USB_INTRUSB (USB common interrupts) register, 26-107
USB_INTx_R (route USB/VBUS IRQ to INTx) bits, 26-102
USB_LINKINFO (USB link info) register, 26-137
USB link info (USB_LINKINFO) register, 26-137
USB low-speed EOF 1 (USB_LS_EOF1) register, 26-139
USB_LS_EOF1 (USB low-speed EOF 1) register, 26-139
USB max Rx data in frame (MAX_PACKET_SIZE_R) bits, 26-120
USB max Tx data in frame (MAX_PACKET_SIZE_T) bits, 26-110
USB_NAKLIMIT0 (USB NAK limit 0) register, 26-128
USB NAK limit 0 (USB_NAKLIMIT0) register, 26-128
USB or non-USB part (USBPARTB1V) bit, 26-140
USB OTG
  DMA master channels, 26-87
  features, 26-2
  host negotiation /configuration, 26-80
  interface pins, 26-53
  OTG session request, 26-78
  peripheral mode operation, 26-12
  transferring packets using DMA, 26-90
USB_OTG_DEV_CTL (USB OTG device control) register, 26-132, 26-134
USB OTG device control (USB_OTG_DEV_CTL) register, 26-132, 26-134
USB_OTG_VBUS_MASK (USB OTG VBUS mask) register, 26-136
USB OTG VBUS mask (USB_OTG_VBUS_MASK) register, 26-136
USBPARTB1V (USB part or non-USB part) bit, 26-140
USB peripheral device address (FUNCTION_ADDRESS) bits, 26-100
USB PLL OSC control (USB_PLLOSC_CTRL) register, 26-141

USB_PLLOSC_CTRL (USB PLL OSC control) register, 26-141
USB power management (USB_POWER) register, 26-98
USB_POWER (USB power management) register, 26-98
USB pu/pd restore control (CSR_RSTD) bit, 26-140
USB received byte count in EP0 FIFO (USB_COUNT0) register, 26-126
USB receive interrupt enable (USB_INTRRXE) register, 26-106
USB receive interrupt (USB_INTRRX) register, 26-104
USB reset (RESET) bit, 26-98
USB Rx byte count (RX_COUNT) bits, 26-127
USB Rx byte count (USB_RXCOUNT) register, 26-127
USB Rx control/status EPx (USB_RXCSR) register, 26-121
USB_RXCOUNT (USB Rx byte count) register, 26-127
USB_RXCSR (USB Rx control/status EPx) register, 26-121
USB Rx endpoint x interrupt enable (EPx_RX_E) bits, 26-106
USB Rx endpoint x interrupt (EPx_RX) bits, 26-104
USB_RXINTERVAL (USB Rx interval) register, 26-130
USB Rx interval (USB_RXINTERVAL) register, 26-130
USB_RX_MAX_PACKET (USB Rx max packet) register, 26-120
USB Rx max packet (USB_RX_MAX_PACKET) register, 26-120
USB Rx poll interval (RX_POLL_INTERVAL) bits, 26-130
USB_RXTYPE (USB Rx type) register, 26-129
USB Rx type (USB_RXTYPE) register, 26-129
USB_SRP_CLKDIV (USB SRP clock divider) register, 26-142
USB SRP clock divider (USB_SRP_CLKDIV) register, 26-142
USB transmit interrupt enable (USB_INTRTXE) register, 26-105
USB transmit interrupt (USB_INTRTX) register, 26-103
USB Tx byte count (TX_COUNT) bits, 26-131
USB Tx byte count (USB_TXCOUNT) register, 26-131
USB Tx control/status EPx (USB_TXCSR) register, 26-115
USB_TXCOUNT (USB Tx byte count) register, 26-131
USB_TXCSR (USB Tx control/status EPx) register, 26-115
USB Tx endpoint x interrupt enable (EPx_TX_E) bits, 26-105
USB Tx endpoint x interrupt (EPx_TX) bits, 26-103
USB_TXINTERVAL (USB Tx interval) register, 26-128
USB Tx interval (USB_TXINTERVAL) register, 26-128
USB_TX_MAX_PACKET (USB Tx max packet) register, 26-110
USB Tx max packet (USB_TX_MAX_PACKET) register, 26-110

USB Tx poll interval
(TX_POLL_INTERVAL) bits, 26-128
USB_TXTYPE (USB Tx type) register, 26-127
USB Tx type (USB_TXTYPE) register, 26-127
USB VBUS pulse length (USB_VPLEN) register, 26-137
USB_VPLEN (USB VBUS pulse length) register, 26-137
user mode, 17-10
UTE (urgency threshold enable) bit, 6-41
UTHE[15:0] field, 6-90

V

VBUS1–0 (VBUS level indicator) bit, 26-132, 26-134
VBUS_ERROR_BE (VBus threshold IRQ enable) bit, 26-108
VBUS_ERROR_B (VBus threshold indicator) bit, 26-107
VBUS level indicator (VBUS1–0) bit, 26-132, 26-134
VBUS pulse length (VPLEN) bits, 26-137
VBus threshold indicator (VBUS_ERROR_B) bit, 26-107
VBus threshold IRQ enable (VBUS_ERROR_BE) bit, 26-108
VCO, multiplication factors, 18-4
VCO signal, 18-1
VDDEXT pins, 19-9
VDDINT pins, 19-9
VDK, 1-30
vertical blanking, 15-6
vertical blanking interval only submode, 15-10
video frame partitioning, 15-7
video streams
  CIF, 15-8
  NTSC, 15-5
  PAL, 15-5
  QCIF, 15-8
VisualDSP++, 17-21
  debugger, 1-30
  development environment, 1-29
VLAN1TAG[15:0] field, 21-77
VLAN2TAG[15:0] field, 21-78
VLEV[3:0] field, 18-20
voltage, 18-16
  changing, 18-19
  control, 18-7
  dynamic control, 18-16
  level values, 18-20
voltage controlled oscillator (VCO), 18-3
voltage regulator, 1-27
voltage regulator controller, 18-19
voltage regulator control (VR_CTL) register, 18-19, 18-25, 18-29, 18-30
VPLEN (VBUS pulse length) bits, 26-137
VR_CTL (voltage regulator control) register, 18-19, 18-25, 18-29, 18-30

W

W1C operations, 6-11
wait for connect (WTCON) bits, 26-137
wait from IDPULLUP (WTID) bits, 26-137
wait states, additional, 7-15
WAKE bit, 18-29, 18-30
WAKEDET (wake-up detected status) bit, 21-35, 21-92, 21-93
wake from hibernate, MAC, 21-30
wake from sleep, MAC, 21-31
wakeup filter 0 address type bit, 21-86, 21-87
wakeup filter 0 pattern CRC[15:0] field, 21-89

wakeup filter 0 pattern offset[7:0] field, 21-88
wakeup filter 1 address type bit, 21-86, 21-87
wakeup filter 1 pattern CRC[15:0] field, 21-89
wakeup filter 1 pattern offset[7:0] field, 21-88
wakeup filter 2 address type bit, 21-86
wakeup filter 2 pattern CRC[15:0] field, 21-89
wakeup filter 2 pattern offset[7:0] field, 21-88
wakeup filter 3 address type bit, 21-86
wakeup filter 3 pattern CRC[15:0] field, 21-89
wakeup filter 3 pattern offset[7:0] field, 21-88
wakeup frame detected, 21-41
wakeup function, 5-7
watchdog control (WDOG_CTL) register, 12-7, 12-8
watchdog count[15:0] field, 12-6
watchdog count[31:16] field, 12-6
watchdog count (WDOG_CNT) register, 12-5, 12-6
watchdog status[15:0] field, 12-7
watchdog status[31:16] field, 12-7
watchdog status (WDOG_STAT) register, 12-3, 12-4, 12-6, 12-7
watchdog timer, 1-24, 12-1 to 12-10
  block diagram, 12-3
  disabling, 12-5
  and emulation mode, 12-2
  enabling with zero value, 12-5
  features, 12-2
  internal interface, 12-3
  overview, 1-24
watchdog timer, 1-24, 12-1 to 12-10 *(continued)*
  registers, 12-5
  reset, 12-5, 17-5, 17-7
  starting, 12-4
waveform generation, pulse width modulation, 10-13
WB_EDGE (write buffer edge detect) bit, 20-17, 20-18
WB_FULL (write buffer full) bit, 20-17
WB_OVF (write buffer overflow) bit, 20-18
WDEN[7:0] field, 12-7
WDEV[1:0] field, 12-4, 12-7
WDOG_CNT (watchdog count) register, 12-5, 12-6
WDOG_CTL (watchdog control) register, 12-7, 12-8
WDOG_STAT (watchdog status) register, 12-3, 12-4, 12-6, 12-7
WDRESET, 16-47, 16-48, 17-102
WDSIZE[1:0] field, 6-69, 6-72
WDTH_CAP mode, 10-23, 10-43
  control bit and register usage, 10-46
WLS[1:0] field, 25-22
WNR bit, 6-72
WNR (DMA direction) bit, 6-69, 6-72, 8-6
WOFF[9:0] field, 24-22, 24-67
WOM (write open drain master) bit, 22-15, 22-37
word length
  SPI, 22-36
  SPORT, 24-28
  SPORT receive data, 24-62
  SPORT transmit data, 24-59